

A Linguagem de Programação Go

Alan A. A. Donovan
Brian W. Kernighan



novatec



A Linguagem de Programação Go

Alan A. A. Donovan

Google Inc.

Brian W. Kernighan

Princeton University

Tradução

Lúcia A. Kinoshita

Revisão técnica

Luciano Ramalho (ThoughtWorks, Inc.)



Addison
Wesley

Novatec

Authorized translation from the English language edition, entitled GO PROGRAMMING LANGUAGE, THE, 1st Edition, by ALAN DONOVAN; BRIAN KERNIGHAN, published by Pearson Education, Inc, publishing as Addison-Wesley Professional, Copyright © 2017 by Alan A. A. Donovan & Brian W. Kernighan.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc. PORTUGUESE language edition published by NOVATEC EDITORA LTDA., Copyright © 2017.

Tradução autorizada da edição original em inglês, intitulada GO PROGRAMMING LANGUAGE, THE, 1st Edition, por ALAN DONOVAN; BRIAN KERNIGHAN, publicada pela Pearson Education, Inc, publicando como Addison-Wesley Professional, Copyright © 2017 por Alan A. A. Donovan & Brian W. Kernighan.

Todos os direitos reservados. Nenhuma parte deste livro pode ser reproduzida ou transmitida por qualquer forma ou meio, eletrônica ou mecânica, incluindo fotocópia, gravação ou qualquer sistema de armazenamento de informação, sem a permissão da Pearson Education, Inc. Edição em Português publicada pela NOVATEC EDITORA LTDA., Copyright © 2017.

Todos os direitos reservados e protegidos pela Lei 9610 de 19/02/1998. É proibida a reprodução desta obra, mesmo parcial, por qualquer processo, sem prévia autorização, por escrito, do autor e da Editora.

Editor: Rubens Prates

Assistente editorial: Priscila A. Yoshimatsu

Tradução: Lúcia A. Kinoshita

Revisão técnica: Luciano Ramalho (ThoughtWorks, Inc.)

Editores eletrônicos: Carolina Kuwabata

Revisão gramatical: Smirna Cavaleiro/Solange Martins

ISBN: 978-85-7522-655-1

Histórico de edições impressas:

Março/2017 Primeira edição

Novatec Editora Ltda.

Rua Luís Antônio dos Santos 110

02460-000 – São Paulo, SP – Brasil

Tel.: +55 11 2959-6529

Email: novatec@novatec.com.br

Site: www.novatec.com.br

Twitter: twitter.com/novateceditora

Facebook: facebook.com/novatec

LinkedIn: linkedin.com/in/novatec

Para Leila e Meg

Sumário

Prefácio

[Origens de Go](#)

[Projeto Go](#)

[Como este livro está organizado](#)

[Onde encontrar mais informações](#)

[Agradecimentos](#)

1 ■ Tutorial

[1.1 Hello, World](#)

[1.2 Argumentos de linha de comando](#)

[1.3 Encontrando linhas duplicadas](#)

[1.4 GIFs animados](#)

[1.5 Buscando um URL](#)

[1.6 Buscando URLs de modo concorrente](#)

[1.7 Um servidor web](#)

[1.8 Miscelâneas](#)

2 ■ Estrutura dos programas

[2.1 Nomes](#)

[2.2 Declarações](#)

[2.3 Variáveis](#)

[2.3.1 Declarações curtas de variáveis](#)

[2.3.2 Ponteiros](#)

[2.3.3 A função new](#)

[2.3.4 Tempo de vida das variáveis](#)

[2.4 Atribuições](#)

[2.4.1 Atribuição de tupla](#)

[2.4.2 Possibilidade de atribuição](#)

[2.5 Declarações de tipos](#)

[2.6 Pacotes e arquivos](#)

[2.6.1 Importações](#)

[2.6.2 Inicialização de pacotes](#)

[2.7 Escopo](#)

3 ■ Tipos de dados básicos

[3.1 Inteiros](#)

[3.2 Números de ponto flutuante](#)

[3.3 Números complexos](#)

[3.4 Booleanos](#)

[3.5 Strings](#)

[3.5.1 Strings literais](#)

[3.5.2 Unicode](#)

[3.5.3 UTF-8](#)

[3.5.4 Strings e fatias de bytes](#)

[3.5.5 Conversões entre strings e números](#)

[3.6 Constantes](#)

[3.6.1 Gerador de constantes iota](#)

[3.6.2 Constantes sem tipo](#)

4 ■ Tipos compostos

[4.1 Arrays](#)

[4.2 Fatias](#)

[4.2.1 Função append](#)

[4.2.2 Técnicas in-place para fatias](#)

[4.3 Mapas](#)

[4.4 Estruturas](#)

[4.4.1 Estruturas literais](#)

[4.4.2 Comparando estruturas](#)

[4.4.3 Inclusão de estruturas e campos anônimos](#)

[4.5 JSON](#)

[4.6 Templates de texto e HTML](#)

5 ■ Funções

[5.1 Declarações de funções](#)

[5.2 Recursão](#)

[5.3 Múltiplos valores de retorno](#)

5.4 Erros

5.4.1 Estratégias de tratamento de erros

5.4.2 Fim de arquivo (EOF)

5.5 Valores função

5.6 Funções anônimas

5.6.1 Cuidado: captura de variáveis de iteração

5.7 Funções variádicas

5.8 Chamadas de função adiadas

5.9 Pânico

5.10 Recuperação

6 ■ Métodos

6.1 Declarações de métodos

6.2 Métodos cujo receptor é um ponteiro

6.2.1 Nil é um valor válido de receptor

6.3 Compondo tipos por meio de inclusão de estruturas

6.4 Valores e expressões método

6.5 Exemplo: tipo vetor de bits

6.6 Encapsulamento

7 ■ Interfaces

7.1 Interfaces como contratos

7.2 Tipos interface

7.3 Como satisfazer uma interface

7.4 Fazendo parse de flags com flag.Value

7.5 Valores interface

7.5.1 Ressalva: uma interface contendo um ponteiro nil não é nil

7.6 Ordenação com sort.Interface

7.7 A interface http.Handler

7.8 Interface error

7.9 Exemplo: avaliador de expressões

7.10 Asserções de tipo

7.11 Diferenciando erros com asserções de tipo

7.12 Consultando comportamentos com asserções de tipo interface

7.13 Switches de tipo

[7.14 Exemplo: decodificação de XML baseada em token](#)

[7.15 Alguns conselhos](#)

8 ■ Gorrotinas e canais

[8.1 Gorrotinas](#)

[8.2 Exemplo: Servidor de relógio concorrente](#)

[8.3 Exemplo: Servidor de eco concorrente](#)

[8.4 Canais](#)

[8.4.1 Canais sem buffer](#)

[8.4.2 Pipelines](#)

[8.4.3 Canais unidirecionais](#)

[8.4.4 Canais com buffer](#)

[8.5 Looping em paralelo](#)

[8.6 Exemplo: Web crawler concorrente](#)

[8.7 Multiplexando com select](#)

[8.8 Exemplo: Travessia concorrente de diretório](#)

[8.9 Cancelamento](#)

[8.10 Exemplo: Servidor de bate-papo](#)

9 ■ Concorrência com variáveis compartilhadas

[9.1 Condições de concorrência](#)

[9.2 Exclusão mútua: sync.Mutex](#)

[9.3 Mutexes de leitura/escrita: sync.RWMutex](#)

[9.4 Sincronização de memória](#)

[9.5 Inicialização lazy: sync.Once](#)

[9.6 O detector de concorrência](#)

[9.7 Exemplo: Cache concorrente não bloqueante](#)

[9.8 Gorrotinas e threads](#)

[9.8.1 Pilhas que podem aumentar](#)

[9.8.2 Escalonamento de gorrotinas](#)

[9.8.3 GOMAXPROCS](#)

[9.8.4 Gorrotinas não têm identidade](#)

10 ■ Pacotes e a ferramenta go

[10.1 Introdução](#)

[10.2 Caminhos de importação](#)

- [10.3 A declaração do pacote](#)
- [10.4 Declarações de importação](#)
- [10.5 Importações vazias](#)
- [10.6 Pacotes e nomenclatura](#)
- [10.7 Ferramenta go](#)
 - [10.7.1 Organização do workspace](#)
 - [10.7.2 Fazendo download de pacotes](#)
 - [10.7.3 Build de pacotes](#)
 - [10.7.4 Documentando pacotes](#)
 - [10.7.5 Pacotes internos](#)
 - [10.7.6 Consulta de pacotes](#)

11 ■ Testes

- [11.1 A ferramenta go test](#)
- [11.2 Funções Test](#)
 - [11.2.1 Testes aleatórios](#)
 - [11.2.2 Testando um comando](#)
 - [11.2.3 Testes caixa-branca](#)
 - [11.2.4 Pacotes de testes externos](#)
 - [11.2.5 Escrevendo testes eficazes](#)
 - [11.2.6 Evitando testes frágeis](#)
- [11.3 Cobertura](#)
- [11.4 Funções Benchmark](#)
- [11.5 Profiling](#)
- [11.6 Funções Example](#)

12 ■ Reflexão

- [12.1 Por que usar reflexão?](#)
- [12.2 reflect.Type e reflect.Value](#)
- [12.3 Display: um printer recursivo de valores](#)
- [12.4 Exemplo: Codificando expressões-S](#)
- [12.5 Atualizando variáveis com reflect.Value](#)
- [12.6 Exemplo: Decodificando expressões-S](#)
- [12.7 Acessando tags de campos de estrutura](#)
- [12.8 Exibindo os métodos de um tipo](#)

[12.9 Uma advertência](#)

[13 ■ Programação de baixo nível](#)

[13.1 unsafe.Sizeof, Alignof e Offsetof](#)

[13.2 unsafe.Pointer](#)

[13.3 Exemplo: Equivalência profunda](#)

[13.4 Chamando código C com cgo](#)

[13.5 Outra advertência](#)

Prefácio

“Go é uma linguagem de programação de código aberto que facilita a criação de softwares simples, confiáveis e eficientes.” (Do site de Go em golang.org)

A linguagem Go foi criada em setembro de 2007 por Robert Griesemer, Rob Pike e Ken Thompson, todos do Google, e foi lançada em novembro de 2009. Os objetivos da linguagem e das ferramentas que a acompanham eram ser expressiva, ser eficiente tanto para compilar quanto para executar e eficaz para escrever programas confiáveis e robustos.

Go ostenta uma semelhança superficial com C e, como ela, é uma ferramenta para programadores profissionais, alcançando o máximo de efeito com o mínimo de recursos. Porém, Go é muito mais que uma versão atualizada de C. Ela empresta e adapta boas ideias de várias outras linguagens, ao mesmo tempo que evita funcionalidades que resultaram em complexidade e em códigos não confiáveis. Seus recursos para concorrência são novos e eficientes, e sua abordagem para abstração de dados e programação orientada a objetos é surpreendentemente flexível. Ela tem gerenciamento de memória automático, isto é, *coleta de lixo* (garbage collection).

Em especial, Go é bem apropriada para criação de infraestruturas como servidores em rede e ferramentas e sistemas para programadores, mas ela é realmente uma linguagem de propósito geral; Go pode ser usada em domínios tão diversos quanto na área gráfica, em aplicações móveis e em aprendizagem automática (machine learning). Ela se tornou popular como substituta de linguagens de scripting não tipadas, pois equilibra expressividade e proteção: programas Go normalmente executam mais rápido que programas escritos em linguagens dinâmicas e sofrem muito menos falhas por causa de erros inesperados de tipagem.

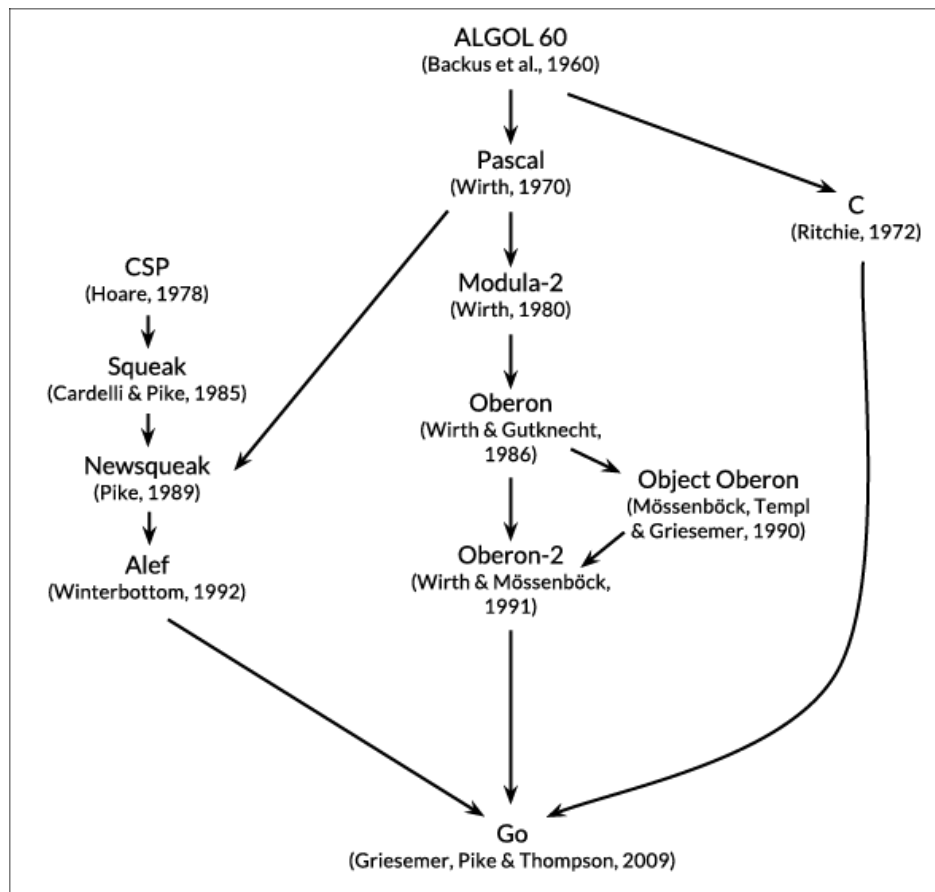
Go é um projeto de código aberto e, sendo assim, o código-fonte de seu compilador, das bibliotecas e das ferramentas está disponível gratuitamente a todos. Contribuições ao projeto são feitas por uma comunidade ativa em todo o mundo. Go executa em sistemas do tipo Unix – Linux, FreeBSD, OpenBSD, Mac OS X –, no Plan 9 e no Microsoft Windows. Programas escritos em um desses ambientes geralmente funcionam sem modificação nos demais.

O propósito deste livro é ajudar você a começar a usar a linguagem Go imediatamente, e usá-la bem, de modo eficaz, aproveitando todas as vantagens de seus recursos e de suas bibliotecas-padrão para escrever programas claros, idiomáticos e eficientes.

Origens de Go

Assim como as espécies biológicas, linguagens de sucesso dão origem a descendentes que incorporam as vantagens de seus ancestrais; às vezes, cruzamentos resultam em pontos surpreendentemente fortes e, ocasionalmente, um novo recurso radical sem precedentes aparece. Podemos aprender muito sobre o motivo pelo qual uma linguagem é como é e para que ambiente ela foi adaptada ao observarmos essas influências.

A figura a seguir mostra as principais influências de linguagens de programação anteriores no design de Go.



Às vezes, Go é descrita como uma “linguagem do tipo C” ou como “C para o século XXI”. De C, Go herdou a sintaxe de suas expressões, as instruções de controle de fluxo, tipos básicos de dados, passagem de parâmetro por valor, ponteiros e, acima de tudo, a ênfase de C em programas compilados que geram código de máquina eficiente e cooperem naturalmente com as abstrações dos sistemas operacionais atuais.

Contudo, há outros ancestrais na árvore genealógica de Go. Uma linha principal de influência vem das linguagens de Niklaus Wirth, começando com Pascal. Modula-2 inspirou o conceito de pacote. Oberon eliminou a distinção entre arquivos de interface de módulos e arquivos de implementação de módulos. Oberon-2 influenciou a sintaxe de pacotes, importações e declarações, e Object Oberon forneceu a sintaxe para declarações de métodos.

Outra linhagem entre os ancestrais de Go, e uma que distingue Go das

linguagens de programação recentes, é uma sequência pouco conhecida de linguagens de pesquisa, desenvolvida no Bell Labs, todas inspiradas no conceito de CSP (*communicating sequential processes*, ou processos sequenciais comunicantes), do artigo inspirador de Tony Hoare, de 1978, sobre os fundamentos da concorrência. Em CSP, um programa é uma composição paralela de processos que não têm estados compartilhados; os processos se comunicam e se sincronizam usando canais. Porém, o CSP de Hoare era uma linguagem formal para descrever os conceitos fundamentais de concorrência, e não uma linguagem de programação para escrever programas executáveis.

Rob Pike e outras pessoas começaram a realizar testes com implementações de CSP como linguagens de verdade. A primeira recebeu o nome de Squeak (“A language for communicating with mice”¹), que era uma linguagem para tratar eventos de mouse e de teclado, com canais criados estaticamente. A essa linguagem seguiu-se a Newsqueak, que oferecia instruções e sintaxe de expressões semelhantes a C e notação de tipos semelhante a Pascal. Era uma linguagem puramente funcional com coleta de lixo, novamente com o propósito de administrar eventos de teclado, mouse e janelas. Os canais tornaram-se valores de primeira classe, criados dinamicamente e armazenáveis em variáveis.

O sistema operacional Plan 9 levou essas ideias adiante em uma linguagem chamada Alef. A linguagem Alef tentou transformar a Newsqueak em uma linguagem de programação de sistemas viável, mas a ausência da coleta de lixo automática dificultou muito a concorrência.

Outras construções em Go mostram a influência de genes não ancestrais aqui e ali; por exemplo, **iota**, de modo geral, é proveniente de APL, e o escopo léxico com funções aninhadas vem de Scheme (e da maioria das linguagens desde então). Também encontramos novas mutações. As fatias (slices) inovadoras de Go oferecem arrays dinâmicos com acesso aleatório eficiente, mas também possibilitam arranjos sofisticados de compartilhamento de memória que lembram listas ligadas. A instrução **defer** surgiu com Go.

Projeto Go

Todas as linguagens de programação refletem a filosofia de programação de seus criadores, o que, frequentemente, inclui um componente significativo de reação às deficiências percebidas em linguagens anteriores. O projeto Go nasceu da frustração com vários sistemas de software no Google, que sofriam de uma explosão de complexidade. (Esse problema, de forma alguma, é exclusivo do Google.)

Como afirma Rob Pike, “a complexidade é multiplicativa”: corrigir um problema deixando uma parte do sistema mais complexo, lentamente, mas de forma certa, acrescenta complexidade a outras partes. Com uma pressão constante para acrescentar recursos, opções e configurações, e disponibilizar código rapidamente, é fácil negligenciar a simplicidade, mesmo que, em longo prazo, ela seja a chave para um bom software.

Simplicidade exige mais trabalho no início de um projeto para reduzir uma ideia à sua essência, e mais disciplina durante o tempo de vida de um projeto para diferenciar mudanças boas de mudanças ruins ou nocivas. Com esforço suficiente, uma mudança boa pode ser acomodada sem comprometer o que Fred Brooks chamava de “integridade conceitual” do design, mas não uma mudança ruim, e uma mudança nociva troca a simplicidade por sua prima leviana, a conveniência. Somente por meio da simplicidade do design um sistema pode permanecer estável, seguro e coerente à medida que cresce.

O projeto Go inclui a linguagem em si, suas ferramentas e bibliotecas-padrão e, por último, mas não menos importante, uma proposta cultural de simplicidade radical. Por ser uma linguagem recente de alto nível, Go tem a vantagem de usufruir de experiências anteriores, e o básico foi muito bem-feito: ela tem coleta de lixo, um sistema de pacotes, funções de primeira classe, escopo léxico, uma interface para chamadas de sistema e strings imutáveis em que o texto geralmente é codificado em UTF-8. Porém, ela tem poucos recursos quando comparada a outras linguagens e é pouco provável que vá acrescentar outros. Por exemplo, Go não tem conversões numéricas implícitas, construtores ou destrutores, sobrecarga

de operadores, valores default para parâmetros, herança, genéricos, exceções, macros, anotações de funções nem armazenamento local para threads. A linguagem é madura e estável, e garante retrocompatibilidade: programas Go mais antigos podem ser compilados e executados com versões mais novas de compiladores e bibliotecas-padrão.

Go tem um sistema de tipos suficiente para evitar a maioria dos erros por descuido que aflige programadores de linguagens dinâmicas, porém seu sistema é mais simples que os de linguagens tipadas comparáveis a ela. Essa abordagem, às vezes, pode resultar em regiões isoladas de programação “não tipada” em um framework mais amplo de tipos, e programadores que usam Go não chegam ao ponto de fazer o que programadores de C++ e Haskell fazem para expressar propriedades de segurança como provas baseadas em tipos. Na prática, porém, Go oferece aos programadores uma boa dose de proteção e de vantagens quanto ao desempenho em tempo de execução, resultantes de um sistema de tipagem relativamente forte sem o peso de um sistema complexo.

Go incentiva um conhecimento do design contemporâneo dos sistemas de computadores, particularmente da importância da localidade. Seus tipos de dados embutidos e a maioria das estruturas de dados das bibliotecas são concebidos para funcionar naturalmente sem inicialização explícita ou construtores implícitos, portanto há relativamente pouca alocação de memória e escritas em memória ocultas no código. Os tipos agregados de Go (structs e arrays) armazenam seus elementos diretamente, exigindo menos área de armazenagem e menos alocações e ponteiros indiretos em comparação com linguagens que usam campos indiretos. Como o computador moderno é uma máquina que funciona em paralelo, Go tem recursos de concorrência baseados em CSP, conforme mencionamos. As pilhas de tamanho variável das threads leves de Go – ou *goroutines* – são inicialmente tão pequenas que criar uma gorrotina custa pouco e criar um milhão delas é prático.

A biblioteca-padrão de Go, muitas vezes descrita como uma biblioteca com “pilhas incluídas”, oferece blocos de construção claros e APIs para

E/S, processamento de texto, imagens, criptografia, rede e aplicações distribuídas, com suporte para vários formatos de arquivo e protocolos-padrão. As bibliotecas e ferramentas fazem uso intenso de convenções para reduzir a necessidade de configurações e explicações, simplificando assim a lógica dos programas; isso deixa programas Go diversificados mais semelhantes uns aos outros e, desse modo, mais fáceis de aprender. Projetos compilados com a ferramenta **go** usam apenas nomes de arquivo e de identificadores e, ocasionalmente, um comentário especial para determinar todas as bibliotecas, executáveis, testes, benchmarks, exemplos, variantes específicas de plataforma e documentação para um projeto: o próprio código-fonte em Go contém a especificação para gerar o projeto.

Como este livro está organizado

Supomos que você já programou em uma ou mais linguagens diferentes, sejam elas compiladas como C, C++ e Java, ou interpretadas como Python, Ruby e JavaScript, portanto não explicaremos tudo como faríamos para um completo iniciante. A sintaxe superficial será familiar, assim como as variáveis, constantes, expressões, controle de fluxo e funções.

O capítulo 1 apresenta um tutorial com as construções básicas de Go, por meio de diversos programas para tarefas do cotidiano como ler e escrever em arquivos, formatar texto, criar imagens e comunicar-se com clientes e servidores na internet.

O capítulo 2 descreve os elementos estruturais de um programa Go – declarações, variáveis, tipos novos, pacotes e arquivos, e escopos. O capítulo 3 discute números, booleanos, strings e constantes, além de explicar como o Unicode é processado. O capítulo 4 descreve os tipos compostos, ou seja, tipos formados por tipos mais simples usando arrays, mapas, structs e *fatias* (slices), que é a abordagem de Go para listas dinâmicas. O capítulo 5 aborda funções e discute tratamento de erros, **panic** e **recover** e a instrução **defer**.

Os capítulos de 1 a 5, portanto, contêm o básico, isto é, aspectos que fazem parte do núcleo de qualquer linguagem imperativa. A sintaxe e o estilo de Go às vezes diferem de outras linguagens, mas a maioria dos programadores os compreenderá rapidamente. Os capítulos restantes focam tópicos em que a abordagem de Go é menos convencional: métodos, interfaces, concorrência, pacotes, testes e reflexão (reflection).

Go tem uma abordagem incomum para programação orientada a objetos. Não há hierarquias de classes ou, para dizer a verdade, não há classes; comportamentos de objetos complexos são criados a partir de objetos mais simples por composição, e não por herança. Os métodos podem ser associados a qualquer tipo definido pelo usuário, e não apenas a structs, e o relacionamento entre tipos concretos e tipos abstratos (*interfaces*) é implícito, portanto um tipo concreto pode satisfazer a uma interface desconhecida pelo implementador do tipo. Os métodos são discutidos no capítulo 5, e as interfaces no capítulo 7.

O capítulo 8 apresenta a abordagem de Go para concorrência, que é baseada na ideia de CSP (*communicating sequential processes*, ou processos sequenciais comunicantes), personificada por gorrotinas e canais. O capítulo 9 explica os aspectos mais tradicionais da concorrência baseada em variáveis compartilhadas.

O capítulo 10 descreve os pacotes – o sistema para organizar bibliotecas. Esse capítulo também mostra como usar a ferramenta **go** de modo eficaz; ela permite fazer compilação, testes, benchmarking, formatação de programas, documentação e muitas outras tarefas, tudo em um só comando.

O capítulo 11 trata de testes em relação aos quais Go assume uma abordagem notadamente leve, evitando frameworks carregados de abstrações em favor de bibliotecas e ferramentas simples. As bibliotecas de teste oferecem uma fundação sobre a qual abstrações mais complexas podem ser criadas, se necessário.

O capítulo 12 discute reflexão (reflection), que é a capacidade de um programa analisar sua própria representação durante a execução. A

reflexão é uma ferramenta eficaz, embora deva ser usada com cuidado; este capítulo explica como encontrar o equilíbrio adequado, mostrando como ela é usada para implementar algumas bibliotecas importantes de Go. O capítulo 13 explica os detalhes intrincados da programação de baixo nível que usa o pacote **unsafe** para contornar o sistema de tipos de Go, e quando isso é apropriado.

Cada capítulo tem vários exercícios que podem ser usados para testar sua compreensão de Go e explorar extensões e alternativas aos exemplos do livro.

Todos os exemplos de código do livro, exceto os mais triviais, estão disponíveis para download no repositório público do Git em **gopl.io**. Todo exemplo é identificado pelo seu path de importação do pacote e pode ser convenientemente buscado, compilado e instalado com o comando **go get**. Você precisará escolher um diretório para servir como a área de trabalho (workspace) de Go e definir a variável de ambiente **GOPATH** para que aponte para esse diretório.

A ferramenta **go** criará o diretório, se for necessário. Por exemplo:

```
$ export GOPATH=$HOME/gobook      # escolhe o diretório que será a
                                   # área de trabalho
$ go get gopl.io/ch1/helloworld    # busca, compila, instala
$ $GOPATH/bin/helloworld           # executa
Hello, 世界
```

Para executar os exemplos, você precisará, no mínimo, da versão 1.5 de Go.

```
$ go version
go version go1.5 linux/amd64
```

Siga as instruções em <https://golang.org/doc/install> se a ferramenta **go** em seu computador for mais antiga ou se ela não estiver instalada.

Onde encontrar mais informações

A melhor fonte para obter mais informações sobre Go é o site oficial,

<https://golang.org>, que oferece acesso à documentação, incluindo a *Go Programming Language Specification* (especificação da linguagem de programação Go), aos pacotes-padrão e recursos associados. Há também tutoriais sobre como escrever código em Go, e escrever bem, além de uma ampla variedade de textos online e recursos de vídeos que serão complementos valiosos para este livro. *The Go Blog* em blog.golang.org publica alguns dos melhores textos sobre Go, com artigos sobre o estado da linguagem, planos para o futuro, informações sobre conferências e explicações detalhadas sobre uma ampla variedade de assuntos relacionados a Go.

Um dos aspectos mais úteis do acesso online a Go (e uma limitação lamentável de um livro em papel) é a capacidade de executar programas Go a partir das páginas web que os descrevem. Essa funcionalidade é oferecida pelo Go Playground em play.golang.org, e pode ser incluída em outras páginas, por exemplo, na página inicial em golang.org ou nas páginas de documentação servidas pela ferramenta **godoc**.

O Playground torna conveniente realizar experimentos simples para verificar sua compreensão sobre sintaxe, semântica ou pacotes de biblioteca com programas pequenos; em muitos aspectos, ele assume o lugar de um *REPL* (*read-eval-print loop*, ou laço ler-avaliar-exibir) em outras linguagens. Suas URLs persistentes são ótimas para compartilhar trechos de código em Go com outras pessoas, informar bugs ou fazer sugestões.

Desenvolvido sobre o Playground, o Go Tour em tour.golang.org é uma sequência de pequenas lições interativas sobre as ideias e construções básicas de Go que percorre a linguagem de forma organizada.

A principal deficiência do Playground e do Tour é que eles permitem importar apenas as bibliotecas-padrão, e muitos recursos de biblioteca – rede, por exemplo – estão restritos por motivos práticos ou por questões de segurança. Eles também exigem acesso à internet para compilar e executar cada programa. Portanto, para experimentos mais elaborados, você deverá executar os programas Go em seu próprio computador.

Felizmente, o processo de download é simples; não deve demorar mais que alguns minutos para buscar a distribuição de Go em **golang.org** e começar a escrever e a executar seus próprios programas em Go.

Como Go é um projeto de código aberto, você pode ler o código de qualquer tipo ou função na biblioteca-padrão online em **<https://golang.org/pkg>**; o mesmo código faz parte da distribuição baixada. Use isso para descobrir como algo funciona ou para responder a perguntas sobre detalhes ou simplesmente ver como os especialistas escrevem um código realmente bom em Go.

Agradecimentos

Rob Pike e Russ Cox, integrantes da equipe principal de Go, leram o manuscrito diversas vezes com muita atenção; seus comentários sobre tudo, desde a escolha de palavras à estrutura e organização em geral tiveram valor inestimável. Enquanto preparava a tradução para o japonês, Yoshiki Shibata foi muito além de suas obrigações; seu olhar meticuloso identificou várias inconsistências no texto original e erros no código. Ficamos extremamente satisfeitos com as revisões completas e os comentários críticos em todo o manuscrito, feitos por Brian Goetz, Corey Kosak, Arnold Robbins, Josh Bleacher Snyder e Peter Weinberger.

Somos gratos a Sameer Ajmani, Ittai Balaban, David Crawshaw, Billy Donahue, Jonathan Feinberg, Andrew Gerrand, Robert Griesemer, John Linderman, Minux Ma, Bryan Mills, Bala Natarajan, Cosmos Nicolaou, Paul Staniforth, Nigel Tao e Howard Trickey, pelas muitas sugestões úteis. Agradecemos também a David Brailsford e a Raph Levien, pelos conselhos sobre composição tipográfica, e a Chris Loper, por explicar os muitos mistérios da produção de um e-book.

Nosso editor, Greg Doench da Addison-Wesley, deu o pontapé inicial e tem sido muito prestativo desde então. A equipe de produção da AW – John Fuller, Dayna Isley, Julie Nahil, Chuti Prasertsith e Barbara Wood – tem sido maravilhosa; os autores não poderiam esperar um apoio melhor.

Alan Donovan gostaria de agradecer a Sameer Ajmani, Chris Demetriou,

Walt Drummond e Reid Tatge do Google, por permitirem que ele tivesse tempo para escrever, a Stephen Donovan, pelos seus conselhos e incentivos na hora certa e, acima de tudo, à sua esposa, Leila Kazemi, pelo seu constante entusiasmo e pelo apoio incondicional a este projeto, apesar das longas horas de distração e de ausência da vida familiar que isso implicou.

Brian Kernighan é profundamente grato aos amigos e colegas pela paciência e tolerância enquanto ele se movia lentamente pelo caminho da compreensão e, em especial, à sua esposa, Meg, que o apoia integralmente na escrita de livros e em muito mais.

Nova York
Outubro de 2015

¹ N.T.: Literalmente, quer dizer: “Uma linguagem para se comunicar com ratos”. É um trocadilho, pois, em inglês, o verbo *to squeak* quer dizer guinchar, podendo ser usado para referir-se ao som emitido por ratos; por outro lado, *mice* é o plural de mouse (rato), que se refere também ao dispositivo usado em computadores.

1

Tutorial

Este capítulo faz um tour pelos componentes básicos de Go. Esperamos fornecer informações e exemplos suficientes para você alçar voo e executar tarefas úteis o mais rápido possível. Os exemplos aqui, e na verdade no livro todo, estão voltados a tarefas que talvez você precise fazer no mundo real. Neste capítulo procuramos oferecer uma amostra da diversidade de programas que podem ser escritos em Go, variando desde um processamento simples de arquivos e algumas imagens até clientes e servidores concorrentes na internet. Certamente não explicaremos tudo no primeiro capítulo, mas estudar esses tais programas em uma nova linguagem pode ser uma maneira eficaz de começar.

Ao aprender uma nova linguagem, há uma tendência natural em escrever código como você faria em uma linguagem que já conhece. Tome cuidado com essa postura tendenciosa enquanto aprende Go e tente evitá-la. Procuramos ilustrar e explicar como escrever um bom código em Go, portanto use o código deste livro como guia quando estiver escrevendo seu próprio código.

1.1 Hello, World

Começaremos com o já tradicional exemplo “hello, world”, que aparece no início do livro *The C Programming Language*¹, publicado em 1978. A linguagem C exerceu uma das influências mais diretas sobre Go, e “hello, world” ilustra várias ideias centrais.

gopl.io/ch1/helloworld

```
package main  
  
import "fmt"
```



```
func main() {  
    fmt.Println("Hello, 世界")  
}
```

Go é uma linguagem compilada. O conjunto de ferramentas de Go converte um programa-fonte e todas as suas dependências em instruções na linguagem de máquina nativa de um computador. Essas ferramentas são acessadas por meio de um único comando chamado **go**, que tem vários subcomandos. O mais simples desses subcomandos é **run**, que compila o código-fonte de um ou mais arquivos-fonte cujos nomes terminam com **.go**, faz a ligação com as bibliotecas e então roda o arquivo executável resultante. (Usaremos **\$** como prompt de comandos ao longo do livro.)

```
$ go run helloworld.go
```

Não é nenhuma surpresa que este comando exibe:

```
Hello, 世界
```

Go trata Unicode de modo nativo, portanto é capaz de processar texto em todas as línguas do mundo.

Se o programa for mais que um experimento descartável, é provável que você queira compilá-lo uma vez e salvar o resultado compilado para usar depois. Isso é feito com **go build**:

```
$ go build helloworld.go
```

Esse comando cria um arquivo binário executável chamado **helloworld**, que pode ser executado a qualquer momento, sem processamentos adicionais:

```
$ ./helloworld  
Hello, 世界
```

Identificamos cada exemplo relevante para lembrar que você pode obter seu código-fonte a partir do repositório do livro em **gopl.io**:

```
gopl.io/ch1/helloworld
```

Se executar **go get gopl.io/ch1/helloworld**, esse comando buscará o código-fonte e o colocará no diretório correspondente. Há mais informações sobre esse assunto nas seções 2.6 e 10.7.

Vamos agora falar sobre o programa propriamente dito. Todo código em Go está organizado em pacotes, que são semelhantes a bibliotecas ou módulos em outras linguagens. Um pacote é constituído de um ou mais arquivos-fonte **.go** em um único diretório que define o que o pacote faz. Cada arquivo-fonte começa com uma declaração **package** – nesse caso, **package main** – que define a qual pacote o arquivo pertence, seguida de uma lista de outros pacotes importados e das declarações do programa armazenadas nesse arquivo.

A biblioteca-padrão de Go tem mais de cem pacotes para tarefas comuns como entrada e saída, ordenação e manipulação de texto. Por exemplo, o pacote **fmt** contém funções para exibir saídas formatadas e para processar entradas. **Println** é uma das funções básicas de saída em **fmt**; ela exibe um ou mais valores separados por espaços, com um caractere de quebra de linha no final para que os valores apareçam em uma única linha na saída.

O pacote **main** é especial. Ele define um programa executável independente, e não uma biblioteca. No pacote **main**, a *função main* também é especial – é onde a execução do programa começa. O que **main** faz é o que o programa faz. É claro que **main**, normalmente, chamará funções de outros pacotes para fazer boa parte do trabalho, por exemplo, a função **fmt.Println**.

Devemos dizer ao compilador quais pacotes são necessários ao arquivo-fonte; esse é o papel da declaração **import** que vem depois da declaração **package**. O programa “hello, world” usa apenas uma função de outro pacote, mas a maioria dos programas importará mais pacotes.

Você deve importar exatamente os pacotes de que precisará. Um programa não compilará se houver importações faltando ou se houver importações desnecessárias. Esse requisito rigoroso evita que referências a pacotes não usados se acumulem à medida que os programas evoluem.

As declarações **import** devem estar após a declaração **package**. Depois disso, um programa é constituído de declarações de funções, variáveis, constantes e tipos (introduzidos pelas palavras reservadas **func**, **var**,

const e **type**); na maioria das vezes, a ordem das declarações não importa. Esse programa é tão pequeno quanto possível, pois declara apenas uma função que, por sua vez, chama apenas outra função. Para economizar espaço, por vezes não mostraremos as declarações **package** e **import** quando apresentarmos os exemplos, mas eles estão no arquivo-fonte e devem estar presentes para compilar o código.

Uma declaração de função é constituída da palavra reservada **func**, o nome da função, uma lista de parâmetros (vazia para **main**), uma lista de resultados (também vazia nesse caso) e o corpo da função – as instruções que definem o que ela faz – entre chaves. Daremos uma olhada mais detalhada nas funções no capítulo 5.

Go não exige ponto-e-vírgula no final de instruções ou declarações, exceto quando duas ou mais estiverem na mesma linha. Na verdade, quebras de linha após determinados elementos sintáticos são convertidas em ponto-e-vírgula, portanto o local em que as quebras de linha são colocadas é importante para a análise correta de código em Go. Por exemplo, a chave de abertura { da função deve estar na mesma linha que o final da declaração **func**, e não em uma linha separada; na expressão **x + y**, uma quebra de linha é permitida após o operador **+**, mas não antes.

Go tem uma posição rígida quanto à formatação de código. A ferramenta **gofmt** reescreve código em um formato padrão, e o subcomando **fmt** da ferramenta **go** aplica **gofmt** a todos os arquivos do pacote especificado, ou àqueles que estão no diretório atual, por padrão. Todos os arquivos-fonte em Go do livro foram submetidos a **gofmt**, e você deve criar o hábito de fazer o mesmo com seu próprio código. Estabelecer um formato padrão por decreto elimina muitos debates sem sentido sobre trivialidades e, acima de tudo, permite várias transformações automáticas de código-fonte que seriam impraticáveis se uma formatação arbitrária fosse permitida.

Muitos editores de texto podem ser configurados para executar **gofmt** sempre que você salvar um arquivo, de modo que seu código-fonte estará sempre devidamente formatado. Uma ferramenta relacionada, **goimports**, adicionalmente administra a inserção e a remoção de declarações de

importação conforme for necessário. Ela não faz parte da distribuição padrão, mas você pode obtê-la com este comando:

```
$ go get golang.org/x/tools/cmd/goimports
```

Para a maioria dos usuários, a maneira usual de fazer download e gerar pacotes, executar seus testes, mostrar sua documentação, e assim por diante, é por meio da ferramenta **go**; daremos uma olhada nela na seção 10.7.

1.2 Argumentos de linha de comando

A maioria dos programas processa alguma entrada a fim de produzir alguma saída; essa é, basicamente, a definição de computação. Mas como um programa obtém dados de entrada sobre os quais operará? Alguns programas geram seus próprios dados; com mais frequência, porém, a entrada vem de uma fonte externa: um arquivo, uma conexão de rede, a saída de outro programa, um usuário em um teclado, argumentos de linha de comando ou algo semelhante. Os próximos exemplos discutirão algumas dessas alternativas, começando pelos argumentos de linha de comando.

O pacote **os** oferece funções e outros valores para lidar com o sistema operacional, independentemente de plataforma. Argumentos de linha de comando estão disponíveis a um programa em uma variável chamada **Args**, que faz parte do pacote **os**; assim, seu nome em qualquer lugar fora do pacote **os** é **os.Args**.

A variável **os.Args** é uma *fatia* (slice) de strings. Fatias são uma noção fundamental em Go, e falaremos muito mais sobre elas em breve. Por enquanto, pense em uma fatia como uma sequência **s** de elementos de array dimensionada dinamicamente, em que elementos individuais podem ser acessados como **s[i]** e uma subsequência contígua, como **s[m:n]**. O número de elementos é dado por **len(s)**. Como na maioria das outras linguagens de programação, toda indexação em Go usa intervalos *semiabertos* que incluem o primeiro índice, mas excluem o último, pois isso simplifica a lógica. Por exemplo, a fatia **s[m:n]**, em que 0

$\leq m \leq n \leq \text{len}(s)$, contém $n-m$ elementos.

O primeiro elemento de `os.Args`, `os.Args[0]`, é o nome do comando propriamente dito; os outros elementos são os argumentos apresentados ao programa quando sua execução foi iniciada. Uma expressão de fatia na forma `s[m:n]` produz uma fatia que se refere aos elementos de m a $n-1$, portanto os elementos de que precisamos em nosso próximo exemplo são aqueles que estão na fatia `os.Args[1:len(os.Args)]`. Se m ou n forem omitidos, os defaults serão `0` ou `len(s)` respectivamente, desta forma podemos abreviar a fatia desejada como `os.Args[1:]`.

A seguir, apresentamos uma implementação do comando `echo` do Unix, que exibe seus argumentos de linha de comando em uma única linha. O código importa dois pacotes, especificados como uma lista entre parênteses, e não como declarações `import` individuais. Qualquer uma das formas é permitida, mas, por convenção, a forma de lista é usada. A ordem das importações não importa; a ferramenta `gofmt` organiza os nomes dos pacotes em ordem alfabética. (Quando houver várias versões de um exemplo, geralmente elas serão numeradas para que você possa saber de qual delas estamos falando.)

gopl.io/ch1/echo1

```
// Echo1 exibe seus argumentos de linha de comando.
package main

import (
    "fmt"
    "os"
)

func main() {
    var s, sep string
    for i := 1; i < len(os.Args); i++ {
        s += sep + os.Args[i]
        sep = " "
    }
    fmt.Println(s)
}
```

Comentários começam com `//`. Todo texto a partir de um `//` até o final

da linha é um comentário para os programadores e é ignorado pelo compilador. Por convenção, descrevemos cada pacote em um comentário imediatamente antes de sua declaração de pacote; para um pacote **main** esse comentário é formado por uma ou mais frases completas que descrevem o programa como um todo.

A declaração **var** define duas variáveis, **s** e **sep**, do tipo **string**. Uma variável pode ser inicializada como parte de sua declaração. Se não for inicializada explicitamente, ela o será implicitamente com o *valor zero* para o seu tipo, que é 0 para tipos numéricos e a string vazia "" para strings. Assim, nesse exemplo, a declaração inicializa **s** e **sep** implicitamente com strings vazias. Teremos mais a dizer sobre variáveis e declarações no capítulo 2.

Para números, Go oferece os operadores aritméticos e lógicos usuais. Quando aplicado a strings, porém, o operador + *concatena* os valores, portanto a expressão:

```
sep + os.Args[i]
```

representa a concatenação das strings **sep** e **os.Args[i]**. A instrução que usamos no programa:

```
s += sep + os.Args[i]
```

é uma *instrução de atribuição* que concatena o valor anterior de **s** com **sep** e **os.Args[i]**, e o resultado é atribuído de volta a **s**; é equivalente a:

```
s = s + sep + os.Args[i]
```

O operador += é um *operador de atribuição*. Cada operador aritmético e lógico, como + ou *, tem um operador de atribuição correspondente.

O programa **echo** poderia ter exibido sua saída em um loop, uma parte de cada vez, mas essa versão monta uma string concatenando repetidamente um novo texto no final. A string **s** nasce vazia, ou seja, com valor igual a "", e cada passagem pelo loop acrescenta um texto a ela; após a primeira iteração, um espaço é também inserido de modo que, quando o loop terminar, haverá um espaço entre cada argumento. Esse é um processo quadrático que pode ser custoso se o número de argumentos for grande, mas no caso de **echo**, isso é improvável. Mostraremos algumas versões

melhoradas de **echo** neste capítulo e no próximo, que tratarão de qualquer ineficiência verdadeira.

A variável **i** de índice do loop é declarada na primeira parte do loop **for**. O símbolo **:=** faz parte de uma *declaração curta de variável* (short variable declaration): uma instrução que declara uma ou mais variáveis e lhes fornece tipos apropriados de acordo com os valores dos inicializadores; falaremos mais sobre esse assunto no próximo capítulo.

A instrução de incremento **i++** soma 1 a **i**; é equivalente a **i += 1** que, por sua vez, é equivalente a **i = i + 1**. Há uma instrução de decremento **i--** correspondente que subtrai 1. Elas são instruções, e não expressões, como na maioria das linguagens da família C, portanto **j = i++** não é permitido, e somente a sintaxe posfixa é usada, assim **--i** também não é permitido.

O loop **for** é a única instrução de loop em Go. Ela assume diversas formas, uma das quais é mostrada aqui:

```
for inicialização; condição; pós {  
    // zero ou mais instruções  
}
```

Parênteses jamais são usados em torno dos três componentes de um loop **for**. Porém, as chaves são obrigatórias, e a chave de abertura deve estar na mesma linha que a instrução *pós*.

A instrução opcional de *inicialização* é executada antes de o loop iniciar. Se estiver presente, ela deve ser uma *instrução simples*, isto é, uma declaração curta de variável, um incremento, uma instrução de atribuição ou uma chamada de função. A *condição* é uma expressão booleana que é avaliada no início de cada iteração do loop; se for avaliada como **true**, as instruções controladas pelo loop serão executadas. A instrução *pós* é executada depois do corpo do loop; então, a condição é avaliada novamente. O loop termina quando a condição se torna falsa.

Qualquer uma dessas partes pode ser omitida. Se não houver *inicialização* ou *pós*, os pontos e vírgulas poderão ser omitidos também:

```
// um loop "while" tradicional
```

```
for condição {  
    // ...  
}
```

Se a condição for totalmente omitida em qualquer uma dessas formas, por exemplo em:

```
// um loop infinito tradicional  
for {  
    // ...  
}
```

o loop será infinito, embora loops nessa forma podem ser encerrados de outra maneira, por exemplo, com uma instrução **break** ou **return**.

Outra forma de loop **for** permite iterar sobre um *intervalo* (range) de valores de um tipo de dado como uma string ou uma fatia. Para ilustrar, eis uma segunda versão de **echo**:

gopl.io/ch1/echo2

```
// Echo2 exhibe seus argumentos de linha de comando  
package main  
  
import (  
    "fmt"  
    "os"  
)  
  
func main() {  
    s, sep := "", ""  
    for _, arg := range os.Args[1:] {  
        s += sep + arg  
        sep = " "  
    }  
    fmt.Println(s)  
}
```

Em cada iteração do loop, **range** produz um par de valores: o índice e o valor do elemento nesse índice. Nesse exemplo, não precisamos do índice, mas a sintaxe de um loop **range** exige que, se vamos lidar com o elemento, devemos lidar com o índice também. Uma ideia seria atribuir o índice a uma variável obviamente temporária como **temp** e ignorar seu valor; mas Go não permite ter variáveis locais não usadas, portanto isso resultaria um erro de compilação.

A solução é usar o *identificador vazio*, cujo nome é `_` (isto é, um underscore). O identificador vazio pode ser usado sempre que a sintaxe exigir um nome de variável, mas ela não é necessária à lógica do programa, por exemplo, para descartar um índice indesejado de loop quando precisamos somente do valor do elemento. A maioria dos programadores que usa Go provavelmente usará **range** e `_` para escrever o programa **echo** como fizemos anteriormente, pois a indexação de **os.Args** é implícita, e não explícita, e assim é mais fácil de ser feita corretamente.

Essa versão do programa usa uma declaração curta de variável para declarar e inicializar **s** e **sep**, mas poderíamos igualmente ter declarado as variáveis separadamente. Há várias maneiras de declarar uma variável do tipo string; estas declarações são todas equivalentes:

```
s := ""
var s string
var s = ""
var s string = ""
```

Por que você iria preferir uma forma em detrimento de outra? A primeira forma, uma declaração curta de variável, é a mais compacta, mas pode ser usada somente em uma função e não em variáveis no nível de pacote. A segunda forma conta com a inicialização default com o valor zero para strings, que é `""`. A terceira forma raramente é usada, exceto quando declaramos múltiplas variáveis. A quarta forma é explícita quanto ao tipo da variável, o que é redundante quando ela é do mesmo tipo do valor inicial, mas é necessária em outros casos em que eles não são do mesmo tipo. Na prática, geralmente você usará uma das duas primeiras formas, com uma inicialização explícita para dizer que o valor inicial é importante, e a inicialização implícita para dizer que o valor inicial não importa.

Conforme observado, sempre que executarmos o loop, a string **s** receberá um conteúdo totalmente novo. A instrução `+=` compõe uma nova string concatenando a string anterior, um caractere de espaço e o próximo argumento e, em seguida, atribui a nova string a **s**. O conteúdo anterior

de `s` não é mais usado, portanto será eliminado pela coleta de lixo em seu devido tempo.

Se a quantidade de dados envolvida for grande, isso pode ser custoso. Uma solução mais simples e mais eficiente seria usar a função `Join`, do pacote `strings`:

gopl.io/ch1/echo3

```
func main() {  
    fmt.Println(strings.Join(os.Args[1:], " "))  
}
```

Por fim, se não estivermos interessados em formatação, mas quisermos apenas ver os valores, talvez para depuração, podemos deixar `Println` formatar o resultado:

```
fmt.Println(os.Args[1:])
```

A saída dessa instrução é semelhante àquela que obteríamos com `strings.Join`, mas com colchetes ao redor. Qualquer fatia pode ser exibida dessa maneira.

Exercício 1.1: Modifique o programa `echo` para exibir também `os.Args[0]`, que é o nome do comando que o chamou.

Exercício 1.2: Modifique o programa `echo` para exibir o índice e o valor de cada um de seus argumentos, um por linha.

Exercício 1.3: Experimente medir a diferença de tempo de execução entre nossas versões potencialmente ineficientes e a versão que usa `strings.Join`. (A seção 1.6 mostra parte do pacote `time`, e a seção 11.4 mostra como escrever testes comparativos para uma avaliação sistemática de desempenho.)

1.3 Encontrando linhas duplicadas

Programas para copiar arquivos, exibir, pesquisar, ordenar, contar e realizar atividades afins têm uma estrutura semelhante: um loop pelos dados de entrada, um processamento em cada elemento e a geração de saída durante ou no final da execução. Mostraremos três variações de um

programa chamado **dup**; ele é parcialmente inspirado no comando **uniq** do Unix, que procura linhas duplicadas adjacentes. As estruturas e os pacotes usados são modelos que podem ser facilmente adaptados.

A primeira versão de **dup** exibe todas as linhas que aparecem mais de uma vez na entrada-padrão, precedidas por sua contagem. Esse programa apresenta a instrução **if**, o tipo de dado **map** e o pacote **bufio**.

gopl.io/ch1/dup1

```
// Dup1 exibe o texto de toda linha que aparece mais de
// uma vez na entrada-padrão, precedida por sua contagem.
package main

import (
    "bufio"
    "fmt"
    "os"
)

func main() {
    counts := make(map[string]int)
    input := bufio.NewScanner(os.Stdin)
    for input.Scan() {
        counts[input.Text()]++
    }
    // NOTA: ignorando erros em potencial de input.Err()
    for line, n := range counts {
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        }
    }
}
```

Como ocorre com **for**, os parênteses nunca são usados em torno da condição em uma instrução **if**, mas as chaves são obrigatórias no corpo. Pode haver uma parte **else** opcional, executada se a condição for falsa.

Um *mapa* (map) armazena um conjunto de pares chave/valor e oferece operações para armazenar, recuperar ou testar um item do conjunto, em tempo constante – isto é, seu desempenho independe do número de itens no mapa. A chave pode ser de qualquer tipo cujos valores possam ser comparados com **==**, e as strings são o exemplo mais comum; o valor

pode ser de qualquer tipo. Nesse exemplo, as chaves são strings e os valores são `ints`. A função embutida `make` cria um novo mapa vazio; ela tem outras utilidades também. Os mapas serão discutidos em detalhes na seção 4.3.

Sempre que `dup` lê uma linha de entrada, a linha é usada como uma chave do mapa e o valor correspondente é incrementado. A instrução `counts[input.Text()]++` é equivalente a estas duas instruções:

```
line := input.Text()
counts[line] = counts[line] + 1
```

O fato de o mapa ainda não conter essa chave não é um problema. Na primeira vez em que uma nova linha é vista, a expressão `counts[line]` do lado direito é avaliada com o valor zero de seu tipo, que é 0 para `int`.

Para exibir os resultados, usamos outro loop `for` baseado em `range`, dessa vez, sobre o mapa `counts`. Como antes, cada iteração gera dois resultados: uma chave e o valor do elemento do mapa relacionado a essa chave. A ordem da iteração no mapa não é especificada, mas, na prática, é aleatória, variando de uma execução para outra. Esse design é intencional, pois evita que os programas contem com uma ordem em particular quando nenhuma ordem é garantida.

Vamos falar do pacote `bufio`, que ajuda a deixar a entrada e a saída mais eficientes e mais convenientes. Um de seus recursos mais úteis é um tipo chamado `Scanner` que lê a entrada e separa-a em linhas ou palavras; em geral, é a maneira mais fácil de processar entradas fornecidas naturalmente em linhas.

O programa usa uma declaração curta de variável para criar uma nova variável `input` que se refere a um `bufio.Scanner`:

```
input := bufio.NewScanner(os.Stdin)
```

O scanner lê da entrada-padrão do programa. Cada chamada a `input.Scan()` lê a próxima linha e remove o caractere de quebra de linha do final; o resultado pode ser obtido por meio da chamada a `input.Text()`. A função `Scan` devolve `true` se houver uma linha, e `false` quando não houver mais dados de entrada.

A função `fmt.Printf`, como `printf` em C e em outras linguagens, gera uma saída formatada a partir de uma lista de expressões. Seu primeiro argumento é uma string de formatação que especifica como os argumentos subsequentes devem ser formatados. O formato de cada argumento é determinado por um caractere de conversão – uma letra após um sinal de porcentagem. Por exemplo, `%d` formata um operando inteiro usando notação decimal e `%s` é expandido com o valor de um operando do tipo string.

`Printf` tem mais de uma dúzia dessas conversões que os programadores de Go chamam de *verbos* (verbs). A tabela a seguir está longe de ser uma especificação completa, mas mostra muitos dos recursos disponíveis:

`%d` inteiro em formato decimal

`%x`, `%o`, `%b` inteiro em formato hexadecimal, octal, binário

`%f`, `%g`, `%e` número de ponto flutuante:

`3.141593 3.141592653589793 3.141593e+00`

`%t` booleano: `true` ou `false`

`%c` runa (código Unicode, exibido como caractere)

`%s` string

`%q` string `"abc"` ou runa `'c'` entre aspas

`%v` qualquer valor em um formato natural

`%T` tipo de qualquer valor

`%%` sinal de porcentagem literal (não indica substituição)

A string de formatação em `dup1` também contém uma tabulação `\t` e uma quebra de linha `\n`. Strings literais podem conter esses *caracteres de escape* para representar caracteres que de outro modo seriam invisíveis. `Printf` não escreve uma quebra de linha por padrão. Por convenção, funções de formatação, cujos nomes terminam com `f`, como `log.Printf` e `fmt.Errorf`, usam as regras de formatação de `fmt.Printf`, enquanto aquelas cujos nomes terminam com `ln` assemelham-se a `Println`, formatando seus argumentos como `%v`, seguidos de uma quebra de linha.

Muitos programas leem de sua entrada-padrão, como anteriormente, ou de uma sequência de arquivos nomeados. A próxima versão de **dup** é capaz de ler da entrada-padrão ou pode tratar uma lista de nomes de arquivos usando **os.Open** para abrir cada um deles:

gopl.io/ch1/dup2

```
// Dup2 exibe a contagem e o texto das linhas que aparecem mais de uma
// vez na entrada. Ele lê de stdin ou de uma lista de arquivos nomeados.
package main

import (
    "bufio"
    "fmt"
    "os"
)

func main() {
    counts := make(map[string]int)
    files := os.Args[1:]
    if len(files) == 0 {
        countLines(os.Stdin, counts)
    } else {
        for _, arg := range files {
            f, err := os.Open(arg)
            if err != nil {
                fmt.Fprintf(os.Stderr, "dup2: %v\n", err)
                continue
            }
            countLines(f, counts)
            f.Close()
        }
    }
    for line, n := range counts {
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        }
    }
}

func countLines(f *os.File, counts map[string]int) {
    input := bufio.NewScanner(f)
    for input.Scan() {
        counts[input.Text()]++
    }
}
```

```
    // NOTA: ignorando erros em potencial de input.Err()  
}
```

A função **os.Open** devolve dois valores. O primeiro é um arquivo aberto (***os.File**) usado em leituras subsequentes por **Scanner**.

O segundo resultado de **os.Open** é um valor do tipo embutido **error**. Se **err** for igual ao valor embutido especial **nil**, o arquivo foi aberto com sucesso. O arquivo é lido e, quando o final da entrada é alcançado, **Close** fecha o arquivo e libera todos os recursos. Por outro lado, se **err** for diferente de **nil**, algo deu errado. Nesse caso, o valor do erro descreve o problema. Nosso tratamento de erros simplório exibe uma mensagem no stream de erro-padrão usando **Fprintf** e o verbo **%v**, que exibe um valor de qualquer tipo em formato-padrão e **dup** então continua com o próximo arquivo; a instrução **continue** passa para a próxima iteração do loop **for** externo.

Visando a manter os exemplos de código com um tamanho razoável, nossos exemplos anteriores são, de certo modo, intencionalmente pouco rigorosos quanto ao tratamento de erros. É óbvio que devemos verificar se houve erro em **os.Open**; entretanto, estamos ignorando a possibilidade menos provável de que um erro possa ocorrer enquanto lemos o arquivo com **input.Scan**. Indicaremos lugares em que ignoraremos a verificação de erros, e discutiremos os detalhes sobre tratamento de erros na seção 5.4.

Observe que a chamada a **countLines** antecede a sua declaração. Funções e outras entidades no nível de pacote podem ser declaradas em qualquer ordem.

Um mapa é uma *referência* à estrutura de dados criada por **make**. Quando um mapa é passado para uma função, ela recebe uma cópia da referência, portanto qualquer alteração que a função chamada fizer na estrutura de dados subjacente também será visível pela referência ao mapa por quem fez a chamada. Em nosso exemplo, os valores inseridos no mapa **counts** por **countLines** são vistos por **main**.

As versões anteriores de **dup** funcionam em um modo “streaming”, em

que a entrada é lida e separada em linhas conforme for necessário, portanto, em princípio, esses programas podem lidar com uma quantidade qualquer de dados de entrada. Uma abordagem alternativa é ler toda a entrada na memória em um só grande bloco, separá-la em linhas de uma só vez e então processá-las. A versão a seguir, **dup3**, funciona dessa maneira. Ela introduz a função **ReadFile** (do pacote **io/ioutil**), que lê todo o conteúdo de um arquivo nomeado, e **strings.Split**, que separa uma string em uma fatia de substrings. (**Split** faz o oposto de **strings.Join**, que vimos antes.)

De certo modo, simplificamos **dup3**. Em primeiro lugar, ele só lê arquivos nomeados, e não a entrada-padrão, pois **ReadFile** exige um nome de arquivo como argumento. Em segundo lugar, passamos a contagem das linhas de volta para **main**, pois agora ela é necessária apenas em um lugar.

gopl.io/ch1/dup3

```
package main

import (
    "fmt"
    "io/ioutil"
    "os"
    "strings"
)

func main() {
    counts := make(map[string]int)
    for _, filename := range os.Args[1:] {
        data, err := ioutil.ReadFile(filename)
        if err != nil {
            fmt.Fprintf(os.Stderr, "dup3: %v\n", err)
            continue
        }
        for _, line := range strings.Split(string(data), "\n") {
            counts[line]++
        }
    }
    for line, n := range counts {
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        }
    }
}
```



```
}  
}
```

`ReadFile` devolve uma fatia de bytes que deve ser convertida em uma `string` para que possa ser separada por `strings.Split`. Discutiremos strings e fatias de bytes em detalhes na seção 3.5.4.

Internamente, `bufio.Scanner`, `ioutil.ReadFile` e `ioutil.WriteFile` usam os métodos `Read` e `Write` de `*os.File`, mas é raro que a maioria dos programadores precise acessar essas rotinas de baixo nível diretamente. As funções de alto nível como aquelas de `bufio` e de `io/ioutil` são mais fáceis de usar.

Exercício 1.4: Modifique `dup2` para que exiba os nomes de todos os arquivos em que cada linha duplicada ocorre.

1.4 GIFs animados

O próximo programa mostra o uso básico dos pacotes padrões de imagens de Go, que usaremos para criar uma sequência de imagens bitmap e, em seguida, codificá-la como uma animação GIF.

As imagens, chamadas *figuras de Lissajous*, eram um efeito visual importante em filmes de ficção científica dos anos 60. Elas são as curvas paramétricas geradas por oscilação harmônica em duas dimensões, por exemplo, duas senoides alimentadas nas entradas x e y de um osciloscópio. A figura 1.1 mostra alguns exemplos.

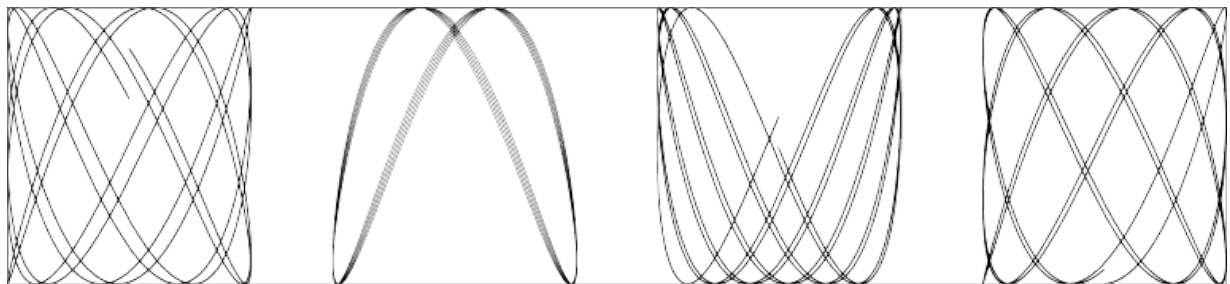


Figura 1.1 – Quatro figuras de Lissajous.

Há várias construções novas neste código, incluindo declarações `const`, estruturas e literais compostos. Diferente da maioria de nossos exemplos,

este envolve também cálculos com números de ponto flutuante. Discutiremos esses assuntos rapidamente aqui, deixando a maior parte dos detalhes para outros capítulos, pois o objetivo principal neste momento é dar uma ideia da aparência de Go e dos tipos de tarefas que podem ser feitos facilmente com a linguagem e suas bibliotecas.

gopl.io/ch1/lissajous

```
// Lissajous gera animações GIF de figuras de Lissajous aleatórias
package main

import (
    "image"
    "image/color"
    "image/gif"
    "io"
    "math"
    "math/rand"
    "os"
    "time"
)

var palette = []color.Color{color.White, color.Black}

const (
    whiteIndex = 0 // primeira cor da paleta
    blackIndex = 1 // próxima cor da paleta
)

func main() {
    rand.Seed(time.Now().UTC().UnixNano())
    lissajous(os.Stdout)
}

func lissajous(out io.Writer) {
    const (
        cycles = 5 // número de revoluções completas do oscilador x
        res     = 0.001 // resolução angular
        size    = 100 // canvas da imagem cobre de [-size...+size]
        nframes = 64 // número de quadros da animação
        delay   = 8 // tempo entre quadros em unidades de 10ms
    )

    freq := rand.Float64() * 3.0 // frequência relativa do oscilador y
    anim := gif.GIF{LoopCount: nframes}
    phase := 0.0 // diferença de fase
    for i := 0; i < nframes; i++ {
        rect := image.Rect(0, 0, 2*size+1, 2*size+1)
```

```

    img := image.NewPaletted(rect, palette)
    for t := 0.0; t < cycles*2*math.Pi; t += res {
        x := math.Sin(t)
        y := math.Sin(t*freq + phase)
        img.SetColorIndex(size+int(x*size+0.5), size+int(y*size+0.5),
            blackIndex)
    }
    phase += 0.1
    anim.Delay = append(anim.Delay, delay)
    anim.Image = append(anim.Image, img)
}
gif.EncodeAll(out, &anim) // NOTA: ignorando erros de codificação
}

```

Após importar um pacote cujo path tem vários componentes, como **image/color**, referenciamos o pacote com um nome proveniente do último componente. Desse modo, a variável **color.White** pertence ao pacote **image/color** e **gif.GIF** pertence a **image/gif**.

Uma declaração **const** (seção 3.6) nomeia constantes, ou seja, valores determinados em tempo de compilação, como os parâmetros numéricos para ciclos, quadros e tempo entre quadros. Assim como as declarações **var**, as declarações **const** podem estar no nível de pacote (portanto os nomes são visíveis em todo o pacote) ou em uma função (os nomes são visíveis somente dentro dessa função). O valor de uma constante deve ser um número, uma string ou um booleano.

As expressões **[]color.Color{...}** e **gif.GIF{...}** são *literais compostos* (seções 4.2 e 4.4.1): uma notação compacta para instanciar qualquer tipo composto de Go a partir de uma sequência de elementos. Nesse caso, o primeiro é uma fatia e o segundo é uma *estrutura* (struct).

O tipo **gif.GIF** é uma estrutura (seção 4.4). Uma estrutura é um grupo de valores chamados *campos* (fields), muitas vezes de tipos diferentes, reunidos em um único objeto que pode ser tratado como uma unidade. A variável **anim** é uma estrutura do tipo **gif.GIF**. A estrutura literal cria um valor de estrutura cujo campo **LoopCount** é definido com **nframes**; todos os outros campos têm o valor zero para seus tipos. Os campos individuais de uma estrutura podem ser acessados com a notação de

ponto, como nas duas últimas atribuições que atualizam explicitamente os campos **Delay** e **Image** de **anim**.

A função **lissajous** tem dois loops aninhados. O loop externo executa 64 iterações, cada qual produzindo um único quadro (frame) da animação. Ele cria uma nova imagem de 201x201 com uma paleta de duas cores, branca e preta. Todos os pixels são inicialmente definidos com o valor zero da paleta (a cor zero da paleta), que definimos com branco. Cada passagem pelo loop interno gera uma nova imagem definindo alguns pixels com preto. O resultado é concatenado a uma lista de frames em **anim**, usando a função embutida **append** (seção 4.2.1), juntamente com um intervalo de tempo especificado de 80ms. Por fim, a sequência de frames e tempos é codificada em formato GIF e escrita na saída padrão **out**. O tipo de **out** é **io.Writer**, que nos permite escrever em uma grande variedade de destinos possíveis, como veremos em breve.

O loop interno executa os dois osciladores. O oscilador x é simplesmente a função seno. O oscilador y também é uma senoidal, mas sua frequência relativa ao oscilador x é um número aleatório entre 0 e 3, e sua fase relativa ao oscilador x é inicialmente zero, mas aumenta a cada frame da animação. O loop executa até o oscilador x completar cinco ciclos completos. A cada passo, ele chama **SetColorIndex** para colorir o pixel correspondente a (x, y) com preto, que está na posição 1 da paleta.

A função **main** chama a função **lissajous**, direcionando-a para escrever na saída-padrão, portanto este comando gera um GIF animado com frames como os da figura 1.1:

```
$ go build gopl.io/ch1/lissajous
$ ./lissajous >out.gif
```

Exercício 1.5: Altere a paleta de cores do programa Lissajous para verde sobre preto, para maior autenticidade. Para criar a cor web **#RRGGBB**, use **color.RGBA{0xRR, 0xGG, 0xBB, 0xff}**, em que cada par de dígitos hexadecimais representa a intensidade do componente vermelho, verde ou azul do pixel.

Exercício 1.6: Modifique o programa Lissajous para gerar imagens em várias

cores adicionando mais valores a **palette** para então exibi-las alterando o terceiro argumento de **SetColorIndex** de alguma maneira interessante.

1.5 Buscando um URL

Para muitas aplicações, acessar informações da Internet é tão importante quanto acessar o sistema de arquivos local. Go oferece uma coleção de pacotes, agrupados como **net**, que facilita enviar e receber informações pela Internet, fazer conexões de baixo nível com a rede e configurar servidores, para os quais os recursos de concorrência de Go (apresentados no capítulo 8) são particularmente úteis.

Para ilustrar o mínimo necessário para recuperar informações via HTTP, a seguir apresentamos um programa simples chamado **fetch**, que busca o conteúdo de cada URL especificado e o exibe como um texto não interpretado; é inspirado no valioso utilitário **curl**. Obviamente, algo mais seria feito com os dados, mas este programa mostra a ideia básica. Ele será usado com frequência neste livro.

gopl.io/ch1/fetch

```
// Fetch exibe o conteúdo encontrado em cada URL especificada.
package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "os"
)

func main() {
    for _, url := range os.Args[1:] {
        resp, err := http.Get(url)
        if err != nil {
            fmt.Fprintf(os.Stderr, "fetch: %v\n", err)
            os.Exit(1)
        }
        b, err := ioutil.ReadAll(resp.Body)
        resp.Body.Close()
        if err != nil {
```

```

        fmt.Fprintf(os.Stderr, "fetch: reading %s: %v\n", url, err)
        os.Exit(1)
    }
    fmt.Printf("%s", b)
}
}

```

Esse programa apresenta funções de dois pacotes: **net/http** e **io/ioutil**. A função **http.Get** faz uma requisição HTTP e, se não houver erro, devolve o resultado na estrutura de resposta **resp**. O corpo **Body** de **resp** contém a resposta do servidor na forma de um stream pronto para leitura. A seguir, **ioutil.ReadAll** lê toda a resposta e o resultado é armazenado em **b**. O stream **Body** é fechado para evitar vazamento de recursos e **Printf** escreve a resposta na saída padrão.

```

$ go build gopl.io/ch1/fetch
$ ./fetch http://gopl.io
<html>
<head>
<title>The Go Programming Language</title>
...

```

Se a requisição HTTP falhar, **fetch** informa a falha:

```

$ ./fetch http://bad.gopl.io
fetch: Get http://bad.gopl.io: dial tcp: lookup bad.gopl.io: no such host

```

Qualquer que seja o caso de erro, **os.Exit(1)** faz o processo sair com um código de status igual a 1.

Exercício 1.7: A chamada de função **io.Copy(dst, src)** lê de **src** e escreve em **dst**. Use-a no lugar de **ioutil.ReadAll** para copiar o corpo da resposta para **os.Stdout** sem exigir um buffer grande o suficiente para armazenar todo o stream. Não se esqueça de verificar o resultado de erro de **io.Copy**.

Exercício 1.8: Modifique **fetch** para que o prefixo **http://** seja acrescentado a cada URL de argumento, caso esteja faltando. Você pode usar **strings.HasPrefix**.

Exercício 1.9: Modifique **fetch** para exibir também o código de status HTTP encontrado em **resp.Status**.

1.6 Buscando URLs de modo concorrente

Um dos aspectos mais interessantes e novos de Go é seu suporte à programação concorrente. É um assunto vasto, e os capítulos 8 e 9 são dedicados a ele; por enquanto, ofereceremos apenas uma amostra dos principais mecanismos de concorrência de Go: gorrotinas e canais.

O próximo programa, **fetchall**, faz a mesma busca do conteúdo de URLs do exemplo anterior, no entanto busca muitos URLs, todos de forma concorrente, de modo que o processo não consumirá mais tempo que a busca mais demorada, em vez de consumir a soma de todos os tempos de busca. Esta versão de **fetchall** descarta as respostas, mas informa o tamanho e o tempo decorrido para cada busca:

gopl.io/ch1/fetchall

```
// Fetchall busca URLs em paralelo e informe os tempos gastos e os tamanhos.
package main

import (
    "fmt"
    "io"
    "io/ioutil"
    "net/http"
    "os"
    "time"
)

func main() {
    start := time.Now()
    ch := make(chan string)
    for _, url := range os.Args[1:] {
        go fetch(url, ch) // inicia uma gorrotina
    }
    for range os.Args[1:] {
        fmt.Println(<-ch) // recebe do canal ch
    }
    fmt.Printf("%.2fs elapsed\n", time.Since(start).Seconds())
}

func fetch(url string, ch chan<- string) {
    start := time.Now()
    resp, err := http.Get(url)
    if err != nil {
```

```

        ch <- fmt.Sprintf(err) // envia para o canal ch
        return
    }
    nbytes, err := io.Copy(ioutil.Discard, resp.Body)
    resp.Body.Close() // evita vazamento de recursos
    if err != nil {
        ch <- fmt.Sprintf("while reading %s: %v", url, err)
        return
    }
    secs := time.Since(start).Seconds()
    ch <- fmt.Sprintf("%.2fs %7d %s", secs, nbytes, url)
}

```

Aqui está um exemplo:

```

$ go build gopl.io/ch1/fetchall
$ ./fetchall https://golang.org http://gopl.io https://godoc.org
0.14s      6852  https://godoc.org
0.16s      7261  https://golang.org
0.48s      2475  http://gopl.io
0.48s elapsed

```

Uma gorrotina (*goroutine*) é uma execução concorrente de função. Um *canal* (channel) é um mecanismo de comunicação que permite que uma gorrotina passe valores de um tipo especificado a outra gorrotina. A função **main** executa em uma gorrotina e a instrução **go** cria gorrotina adicionais.

A função **main** cria um canal de strings usando **make**. Para cada argumento de linha de comando, a instrução **go** no primeiro **for/range** inicia uma nova gorrotina que executa **fetch** assincronamente para buscar o URL usando **http.Get**. A função **io.Copy** lê o corpo da resposta e descarta-o, escrevendo no stream de saída **ioutil.Discard**. **Copy** devolve a contagem de bytes juntamente com qualquer erro ocorrido. À medida que cada resultado chega, **fetch** envia uma linha de resumo pelo canal **ch**. O segundo **for/range** em **main** recebe e exibe essas linhas.

Quando uma gorrotina tenta receber ou enviar em um canal, ela fica bloqueada até outra gorrotina tentar a operação correspondente de envio ou recebimento; quando isso acontece, o valor é transferido e ambas as

gorrotinas continuam. Nesse exemplo, cada **fetch** envia um valor (**ch <- expressão**) pelo canal **ch**, e **main** recebe todos eles (**<-ch**). Deixar todos os **Print** a cargo de **main** garante que a saída de cada gorrotina seja processada como uma unidade, sem o risco de haver uma saída embaralhada, se duas gorrotinas terminarem ao mesmo tempo.

Exercício 1.10: Encontre um site que gere uma grande quantidade de dados. Investigue o caching executando **fetchall** duas vezes sucessivamente para ver se o tempo informado sofre muita alteração. Você sempre obtém o mesmo conteúdo? Modifique **fetchall** para exibir sua saída em um arquivo para que ela possa ser examinada.

Exercício 1.11: Experimente usar **fetchall** com listas mais longas de argumentos, por exemplo, amostras de sites disponíveis em **alexa.com** que fazem parte do primeiro milhão (top million). Como o programa se comporta se um site simplesmente não responder? (A seção 8.9 descreve maneiras de lidar com esses casos.)

1.7 Um servidor web

As bibliotecas de Go facilitam escrever um servidor web que responde a requisições de clientes, como aquelas feitas por **fetch**. Nesta seção mostraremos um servidor mínimo que devolve o componente de path do URL usado para acessar o servidor. Ou seja, se a requisição foi feita para **http://localhost:8000/hello**, a resposta será **URL.Path = "/hello"**.

gopl.io/ch1/server1

```
// Server1 é um servidor de "eco" mínimo.
package main

import (
    "fmt"
    "log"
    "net/http"
)

func main() {
    http.HandleFunc("/", handler) // cada requisição chama handler
    log.Fatal(http.ListenAndServe("localhost:8000", nil))
}
```

```

}
// handler ecoa o componente Path do URL requisitado
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "URL.Path = %q\n", r.URL.Path)
}

```

O programa tem apenas algumas linhas porque as funções de biblioteca fazem a maior parte do trabalho. A função **main** conecta uma função handler a URLs de entrada cujo path comece com /, ou seja, qualquer URL, e inicia um servidor que fica ouvindo requisições de entrada na porta 8000. Uma requisição é representada como uma estrutura do tipo **http.Request**, que contém vários campos relacionados, um dos quais é o URL da requisição de entrada. Quando uma requisição chega, ela é passada para a função **handler**, que extrai o componente path (**/hello**) do URL de requisição e o envia como resposta usando **fmt.Fprintf**. Servidores web serão explicados em detalhes na seção 7.7.

Vamos iniciar o servidor em background. No Mac OS X ou no Linux, acrescente um “e comercial” (&, ou ampersand) ao comando; no Microsoft Windows, execute o comando sem o “e comercial”, em uma janela de comando separada.

```
$ go run src/gopl.io/ch1/server1/main.go &
```

Então podemos fazer requisições de cliente a partir da linha de comando:

```

$ go build gopl.io/ch1/fetch
$ ./fetch http://localhost:8000
URL.Path = "/"
$ ./fetch http://localhost:8000/help
URL.Path = "/help"

```

Ou então, podemos acessar o servidor a partir de um browser, como mostra a figura 1.2.

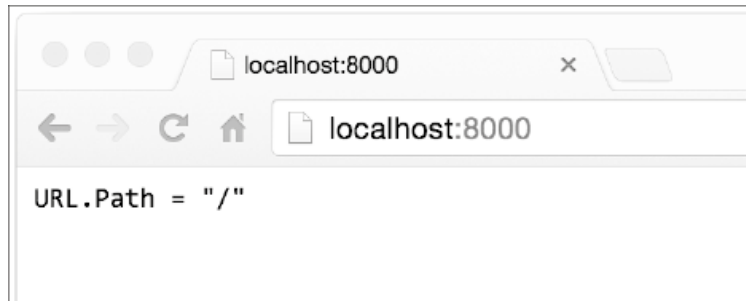


Figura 1.2 – Uma resposta do servidor de eco.

É fácil acrescentar funcionalidades ao servidor. Um acréscimo útil é um URL específico que devolva algum tipo de status. Por exemplo, esta versão executa o mesmo eco, mas também conta o número de requisições; uma requisição ao URL `/count` devolve a contagem até agora, excluindo as próprias requisições `/count`.

gopl.io/ch1/server2

```
// Server2 é um servidor mínimo de "eco" e contador
package main

import (
    "fmt"
    "log"
    "net/http"
    "sync"
)

var mu sync.Mutex
var count int

func main() {
    http.HandleFunc("/", handler)
    http.HandleFunc("/count", counter)
    log.Fatal(http.ListenAndServe("localhost:8000", nil))
}

// handler ecoa o componente Path do URL requisitado
func handler(w http.ResponseWriter, r *http.Request) {
    mu.Lock()
    count++
    mu.Unlock()
    fmt.Fprintf(w, "URL.Path = %q\n", r.URL.Path)
}

// counter ecoa o número de chamadas até agora
```

```
func counter(w http.ResponseWriter, r *http.Request) {
    mu.Lock()
    fmt.Fprintf(w, "Count %d\n", count)
    mu.Unlock()
}
```

O servidor tem dois handlers, e o URL requisitado determina qual deles é chamado: uma requisição para `/count` chama **counter**, e todas as demais chamam **handler**. Um padrão de handler que termine com uma barra corresponde a qualquer URL que tenha o padrão como prefixo. Internamente, o servidor executa o handler para cada requisição de entrada em uma gorrotina separada para que possa servir múltiplas requisições simultaneamente. Entretanto, se duas requisições concorrentes tentarem atualizar **count** ao mesmo tempo, pode ser que ele não seja incrementado consistentemente; o programa teria um bug sério chamado *condição de corrida* (race condition – mais informações na seção 9.1). Para evitar esse problema, devemos garantir que, no máximo, uma gorrotina acesse a variável em determinado instante, que é o propósito das chamadas a **mu.Lock()** e a **mu.Unlock()** em torno de cada acesso a **count**. Daremos uma olhada com mais detalhes na concorrência com variáveis compartilhadas no capítulo 9.

Como um exemplo mais sofisticado, a função handler pode informar os cabeçalhos e os dados de formulário que receber, tornando o servidor útil para inspecionar e depurar requisições:

gopl.io/ch1/server3

```
// handler ecoa a requisição HTTP
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "%s %s %s\n", r.Method, r.URL, r.Proto)
    for k, v := range r.Header {
        fmt.Fprintf(w, "Header[%q] = %q\n", k, v)
    }
    fmt.Fprintf(w, "Host = %q\n", r.Host)
    fmt.Fprintf(w, "RemoteAddr = %q\n", r.RemoteAddr)
    if err := r.ParseForm(); err != nil {
        log.Print(err)
    }
    for k, v := range r.Form {
```

```

        fmt.Fprintf(w, "Form[%q] = %q\n", k, v)
    }
}

```

Esse código usa campos da estrutura **http.Request** para gerar uma saída como esta:

```

GET /?q=query HTTP/1.1
Header["Accept-Encoding"] = ["gzip, deflate, sdch"]
Header["Accept-Language"] = ["en-US,en;q=0.8"]
Header["Connection"] = ["keep-alive"]
Header["Accept"] = ["text/html,application/xhtml+xml,application/xml;..."]
Header["User-Agent"] = ["Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_5)..."]
Host = "localhost:8000"
RemoteAddr = "127.0.0.1:59911"
Form["q"] = ["query"]

```

Observe como a chamada a **ParseForm** está aninhada em uma instrução **if**. Go permite que uma instrução simples, como uma declaração de variável local, anteceda a condição de **if**, o que é particularmente útil para tratamento de erros, como neste exemplo. Poderíamos ter escrito o código assim:

```

err := r.ParseForm()
if err != nil {
    log.Print(err)
}

```

mas combinar as instruções deixa o código mais compacto e reduz o escopo da variável **err**, o que é uma boa prática. Definiremos escopo na seção 2.7.

Nesses programas, vimos três tipos bem diferentes usados como streams de saída. O programa **fetch** copiava os dados da resposta HTTP para **os.Stdout**, um arquivo, como fazia o programa **lissajous**. O programa **fetchall** jogava fora as respostas (enquanto contava seus tamanhos) copiando-as para o sink (consumidor) trivial **ioutil.Discard**. E o servidor web anterior usou **fmt.Fprintf** para escrever em um **http.ResponseWriter** que representa o navegador web.

Embora esses três tipos sejam diferentes quanto aos detalhes sobre o que fazem, todos satisfazem a uma *interface* comum que permite que qualquer

um deles seja usado sempre que um stream de saída seja necessário. Essa interface, chamada `io.Writer`, será discutida na seção 7.1.

O sistema de interface de Go é o assunto do capítulo 7, mas para ter uma ideia do que ele é capaz de fazer, vamos ver como é fácil combinar o servidor web com a função `lissajous` para que os GIFs animados sejam escritos no cliente HTTP, e não na saída-padrão. Basta acrescentar estas linhas no servidor web:

```
handler := func(w http.ResponseWriter, r *http.Request) {  
    lissajous(w)  
}  
http.HandleFunc("/", handler)
```

ou, de modo equivalente:

```
http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {  
    lissajous(w)  
})
```

O segundo argumento da chamada da função `HandleFunc` anterior é uma *função literal*, ou seja, uma função anônima definida no local em que é usada. Explicaremos isso com mais detalhes na seção 5.6.

Feita essa alteração, acesse `http://localhost:8000` em seu navegador. Sempre que carregar a página, você verá uma nova animação, como a da figura 1.3.

Exercício 1.12: Modifique o servidor `Lissajous` para ler valores de parâmetros do URL. Por exemplo, você pode organizá-lo de modo que um URL como `http://localhost:8000/?cycles=20` defina o número de ciclos para 20, em vez de usar o default igual a 5. Utilize a função `strconv.Atoi` para converter o parâmetro do tipo string em um inteiro. Você pode ver a documentação da função usando `go doc strconv.Atoi`.

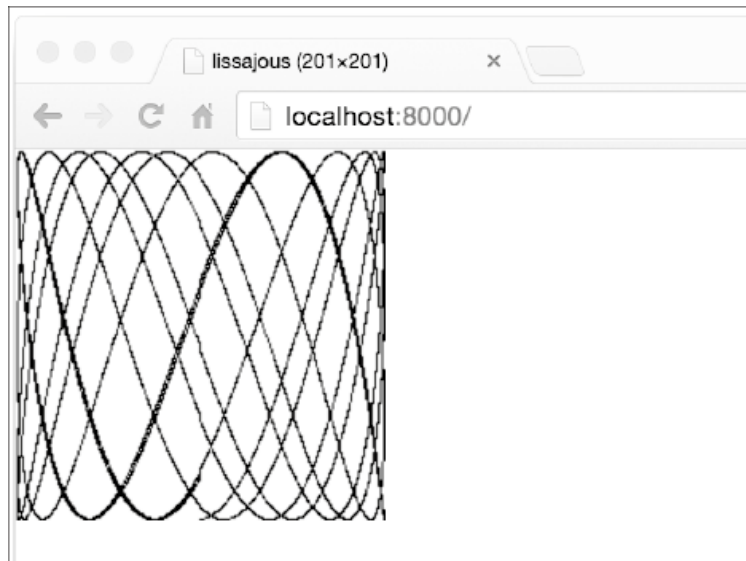


Figura 1.3 – Figuras de Lissajous animadas em um navegador.

1.8 Miscelâneas

Há muito mais sobre Go do que discutimos nesta rápida introdução. A seguir, apresentaremos alguns assuntos que mal abordamos ou que foram totalmente omitidos, com uma discussão suficiente apenas para que sejam familiares quando fizerem aparições rápidas antes de serem discutidos de forma completa.

Controle de fluxo: Discutimos as duas instruções fundamentais de controle de fluxo, **if** e **for**, mas não vimos a instrução **switch**, que oferece múltiplas opções. Eis um pequeno exemplo:

```
switch lançarMoeda() {  
  case "cara":  
    cara++  
  case "coroa":  
    coroa++  
  default:  
    fmt.Println("caiu em pé!")  
}
```

O resultado da chamada a **lançarMoeda** é comparado ao valor de cada caso. Os casos são avaliados de cima para baixo, portanto a primeira correspondência é executada. O caso default opcional será executado se

nenhum dos demais casos corresponderem; ele pode ser colocado em qualquer lugar. Os casos não apresentam continuidade (não fazem fall through) de um para o próximo como nas linguagens do tipo C (embora haja uma instrução **fallthrough**, raramente usada, que introduz esse comportamento).

Um **switch** não precisa de um operando; ele pode simplesmente listar os casos, em que cada um é uma expressão booleana:

```
func Signum(x int) int {  
    switch {  
    case x > 0:  
        return +1  
    default:  
        return 0  
    case x < 0:  
        return -1  
    }  
}
```

Essa forma chama-se *switch sem tag* (tagless switch); é equivalente a **switch true**.

Assim como as instruções **for** e **if**, um **switch** pode incluir uma instrução simples opcional – uma declaração curta de variável, uma instrução de incremento ou de atribuição ou uma chamada de função – que pode ser usada para definir um valor antes de ele ser testado.

As instruções **break** e **continue** modificam o fluxo do controle. Um **break** faz o controle ser retomado na próxima instrução após a instrução **for**, **switch** ou **select** mais interna (que veremos mais adiante) e, como vimos na seção 1.3, um **continue** faz o loop **for** mais interno iniciar sua próxima iteração. As instruções podem ter rótulos para que **break** e **continue** possam se referir a eles, por exemplo, para sair de vários loops aninhados de uma só vez ou iniciar a próxima iteração do loop mais externo. Existe até mesmo uma instrução **goto**, embora seja voltada para código gerado pelo computador, e não para uso normal pelos programadores.

Tipos nomeados: uma declaração **type** permite dar um nome a um tipo

existente. Como tipos referentes a estruturas muitas vezes são longos, quase sempre eles são nomeados. Um exemplo familiar é a definição de um tipo **Point** para um sistema gráfico 2D:

```
type Point struct {  
    X, Y int  
}  
var p Point
```

Declarações de tipo e tipos nomeados serão discutidos no capítulo 2.

Ponteiros: Go oferece ponteiros, isto é, valores que contêm o endereço de uma variável. Em algumas linguagens, notadamente em C, ponteiros são relativamente irrestritos. Em outras linguagens, ponteiros são disfarçados como “referências” e não há muito que se possa fazer com eles a não ser passá-los de um lado para o outro. Go assume uma posição, de certo modo, intermediária. Os ponteiros são explicitamente visíveis. O operador **&** fornece o endereço de uma variável, e o operador ***** recupera a variável à qual o ponteiro se refere, mas não há aritmética com ponteiros. Explicaremos os ponteiros na seção 2.3.2.

Métodos e interfaces: um método é uma função associada a um tipo nomeado; Go é incomum no sentido em que métodos podem ser associados a quase todo tipo nomeado. Os métodos serão discutidos no capítulo 6. As interfaces são tipos abstratos que nos permitem tratar tipos concretos diferentes da mesma maneira, com base nos métodos que eles têm, e não no modo como são representados ou implementados. As interfaces são o assunto do capítulo 7.

Pacotes: Go vem com uma biblioteca-padrão extensa de pacotes úteis, e a comunidade Go vem criando e compartilhando muitos outros. Programação geralmente tem mais a ver com o uso de pacotes existentes que com a escrita de um código original por conta própria. Ao longo do livro destacaremos umas duas dúzias dos pacotes padrões mais importantes, mas há muitos outros que não teremos espaço para mencionar, e não podemos oferecer nada remotamente parecido com uma referência completa a qualquer pacote.

Antes de embarcar em qualquer programa novo, é uma boa ideia ver se já existem pacotes que possam ajudar você a fazer seu trabalho mais facilmente. Você pode encontrar um índice dos pacotes da biblioteca-padrão em <https://golang.org/pkg> e os pacotes resultantes da contribuição da comunidade em <https://godoc.org>. A ferramenta **go doc** deixa esses documentos facilmente acessíveis a partir da linha de comando:

```
$ go doc http.ListenAndServe
package http // import "net/http"

func ListenAndServe(addr string, handler Handler) error
    ListenAndServe listens on the TCP network address addr and then
    calls Serve with handler to handle requests on incoming connections.
...
```

Comentários: já mencionamos comentários para documentação no início de um programa ou de um pacote. Também é considerado um bom estilo escrever um comentário antes da declaração de cada função para especificar o seu comportamento. Essas convenções são importantes porque elas são usadas por ferramentas como **go doc** e **godoc** para localizar e exibir documentação (seção 10.7.4).

Para comentários que se estendam por várias linhas ou que apareçam dentro de uma expressão ou de uma instrução, há também a notação `/* ... */`, conhecida de outras linguagens. Esses comentários às vezes são usados no início de um arquivo para apresentar um bloco grande de texto explicativo, evitando um `//` em cada linha. Em um comentário, `//` e `/*` não têm nenhum significado especial, portanto comentários não são aninhados.

¹ N.T.: Edição brasileira publicada com o título *C: A linguagem de programação - padrão ANSI* (Campus, 1989).

2

Estrutura dos programas

Em Go, assim como em outras linguagens de programação, criamos programas grandes a partir de um conjunto pequeno de construções básicas. Variáveis armazenam valores. Expressões simples são combinadas para formar expressões maiores com operações como adição e subtração. Tipos básicos são reunidos em agregados como arrays e estruturas. Expressões são usadas em instruções cuja ordem de execução é determinada por instruções de controle de fluxo como `if` e `for`. Instruções são agrupadas em funções para isolamento e reutilização. Funções são reunidas em arquivos-fonte e em pacotes.

Vimos exemplos da maior parte disso no capítulo anterior. Neste capítulo apresentaremos mais detalhes sobre os elementos estruturais básicos de um programa em Go. Os programas de exemplo são propositalmente simples para que possamos focar na linguagem, sem nos distrairmos com estruturas de dados ou algoritmos complicados.

2.1 Nomes

Nomes de funções, variáveis, constantes, tipos, rótulos de instruções e pacotes em Go seguem uma regra simples: um nome começa com uma letra (isto é, tudo que o Unicode considera uma letra) ou um underscore, e pode ter qualquer quantidade de letras, dígitos e underscores adicionais. Há distinção entre letras maiúsculas e minúsculas: `heapSort` e `Heapsort` são nomes diferentes.

Go tem 25 *palavras reservadas*, como `if` e `switch`, que podem ser usadas somente onde a sintaxe permite; elas não podem ser utilizadas como nomes.

break	default	func	interface	select
case	defer	go	map	struct
chan	else	goto	package	switch
const	fallthrough	if	range	type
continue	for	import	return	- var

Além disso, há aproximadamente três dúzias de nomes *pré-declarados* como **int** e **true** para constantes, tipos e funções embutidos.

Constantes:	true false iota nil
Tipos:	int int8 int16 int32 int64
	uint uint8 uint16 uint32 uint64 uintptr
	float32 float64 complex128 complex64
	bool byte rune string error
Funções:	make len cap new append copy close delete
	complex real imag
	panic recover

Esses nomes não são reservados, portanto você pode usá-los em declarações. Veremos alguns lugares em que redeclarar um deles pode fazer sentido, mas esteja ciente da possibilidade de confusão.

Se uma entidade for declarada em uma função, ela será *local* a essa função. Porém, se for declarada fora de uma função, ela será visível a todos os arquivos do pacote ao qual ela pertence. O fato de a primeira letra do nome ser maiúscula ou minúscula determina sua visibilidade além das fronteiras dos pacotes. Se o nome começa com uma letra maiúscula, ele é *exportado*, o que significa que é visível e acessível fora de seu próprio pacote e pode ser referenciado por outras partes do programa, como **Printf** do pacote **fmt**. Nomes de pacotes usam sempre letras minúsculas.

Não há limite para o tamanho dos nomes, mas, segundo a convenção e o estilo de programas Go, a tendência é usar nomes curtos, especialmente para variáveis locais com escopos pequenos; é bem mais provável que você veja variáveis chamadas **i** que **theLoopIndex**. Geralmente, quanto maior o escopo de um nome, mais longo e mais significativo ele deve ser.

Do ponto de vista de estilo, programadores de Go usam “camel case” para

compor nomes por combinação de palavras, isto é, letras maiúsculas no interior do nome são preferíveis a underscores no meio. Desse modo, as bibliotecas-padrão têm funções com nomes como `QuoteRuneToASCII` e `parseRequestLine`, mas jamais `quote_rune_to_ASCII` ou `parse_request_line`. As letras de siglas e acrônimos como ASCII e HTML são todas representadas com maiúscula ou minúscula, portanto uma função pode se chamar `htmlEscape`, `HTMLEscape` ou `escapeHTML`, mas nunca `escapeHtml`.

2.2 Declarações

Uma *declaração* dá nome a uma entidade do programa e especifica algumas ou todas as suas propriedades. Há quatro tipos principais de declaração: **var**, **const**, **type** e **func**. Falaremos sobre variáveis e tipos neste capítulo, sobre constantes no capítulo 3 e funções no capítulo 5.

Um programa Go é armazenado em um ou mais arquivos cujos nomes terminam com **.go**. Cada arquivo começa com uma declaração **package** que informa o pacote do qual o arquivo faz parte. A declaração **package** é seguida de qualquer declaração **import** e, em seguida, por uma sequência de declarações de tipos, variáveis, constantes e funções no *nível de pacote* (package-level), em qualquer ordem. Por exemplo, o programa a seguir declara uma constante, uma função e duas variáveis:

gopl.io/ch2/boiling

```
// Boiling exhibe o ponto de ebulição da água.
package main

import "fmt"

const boilingF = 212.0

func main() {
    var f = boilingF
    var c = (f - 32) * 5 / 9
    fmt.Printf("boiling point = %g°F or %g°C\n", f, c)
    // Saída:
    // boiling point = 212°F or 100°C
}
```

A constante **boilingF** é uma declaração de nível de pacote (assim como a função **main**), enquanto as variáveis **f** e **c** são locais à função **main**. O nome de cada entidade no nível de pacote é visível não só por todo o arquivo-fonte que contém sua declaração, mas por todos os arquivos do pacote. Em comparação, declarações locais são visíveis apenas dentro da função em que foram feitas e, às vezes, somente em uma pequena parte dela.

Uma declaração de função tem um nome, uma lista de parâmetros (variáveis cujos valores são fornecidos por quem chama a função), uma lista opcional de resultados e o corpo da função, que contém as instruções que definem o que a função faz. A lista de resultados é omitida se a função não devolve nada. A execução da função começa na primeira instrução e continua até que uma instrução **return** seja encontrada ou o final de uma função que não devolva nenhum resultado seja alcançado. O controle e qualquer resultado são então devolvidos a quem chamou.

Já vimos um número razoável de funções, e há várias outras por vir, incluindo uma discussão ampla no capítulo 5 – o que apresentamos aqui é apenas um esboço. A função **fToC** a seguir encapsula a lógica de conversão de temperatura; ela é definida apenas uma vez, mas pode ser usada a partir de vários lugares. Neste caso, **main** chama a função duas vezes usando os valores de duas constantes locais diferentes:

gopl.io/ch2/ftoc

```
// Ftoc exhibe duas conversões de Fahrenheit para Celsius
package main

import "fmt"

func main() {
    const freezingF, boilingF = 32.0, 212.0
    fmt.Printf("%g°F = %g°C\n", freezingF, fToC(freezingF)) // "32°F = 0°C"
    fmt.Printf("%g°F = %g°C\n", boilingF, fToC(boilingF)) // "212°F = 100°C"
}

func fToC(f float64) float64 {
    return (f - 32) * 5 / 9
}
```

2.3 Variáveis

Uma declaração **var** cria uma variável de um tipo particular, associa um nome a ela e define seu valor inicial. Toda declaração tem o formato geral:

```
var nome tipo = expressão
```

O tipo ou a parte `= expressão` pode ser omitido, mas não ambos. Se o tipo for omitido, ele é determinado pela expressão de inicialização. Se a expressão for omitida, o valor inicial será o *valor zero* do tipo, que é `0` para números, **false** para booleanos, `""` para strings e **nil** para tipos de interfaces e de referência (fatia, ponteiro, mapa, canal, função). O valor zero de um tipo agregado como um array ou uma estrutura corresponde ao valor zero para todos os seus elementos ou campos.

O mecanismo do valor zero garante que uma variável sempre armazena um valor bem definido de seu tipo; em Go não há variáveis não inicializadas. Isso simplifica o código e, com frequência, garante um comportamento sensato para condições singulares, sem trabalho extra. Por exemplo:

```
var s string
fmt.Println(s) // ""
```

exibe uma string vazia, em vez de provocar algum tipo de erro ou apresentar um comportamento imprevisível. Programadores de Go muitas vezes investem algum esforço para deixar o valor zero de um tipo mais complicado ser significativo, de modo que as variáveis comecem suas vidas em um estado utilizável.

É possível declarar e, opcionalmente, inicializar um conjunto de variáveis em uma única declaração, com uma lista de expressões correspondentes. Omitir o tipo permite declarar diversas variáveis de tipos diferentes:

```
var i, j, k int // int, int, int
var b, f, s = true, 2.3, "four" // bool, float64, string
```

Os inicializadores podem ser valores literais ou expressões arbitrárias. Variáveis de nível de pacote são inicializadas antes da função **main** executar (seção 2.6.2), enquanto variáveis locais são inicializadas à medida que suas declarações são encontradas durante a execução de funções.

Um conjunto de variáveis também pode ser inicializado por uma chamada a uma função que devolva diversos valores:

```
var f, err = os.Open(name) // os.Open devolve um arquivo e um erro
```

2.3.1 Declarações curtas de variáveis

Dentro de uma função, uma forma alternativa chamada *declaração curta de variável* (short variable declaration) pode ser usada para declarar e inicializar variáveis locais. Ela assume a forma *nome := expressão*, e o tipo do *nome* é determinado pelo tipo da *expressão*. A seguir, apresentamos três das várias declarações curtas de variáveis da função **lissajous** (seção 1.4):

```
anim := gif.GIF{LoopCount: nframes}  
freq := rand.Float64() * 3.0  
t := 0.0
```

Por serem compactas e flexíveis, declarações curtas de variáveis são usadas para declarar e inicializar a maioria das variáveis locais. Uma declaração **var** tende a ser reservada para variáveis locais que precisem de um tipo explícito diferente daquele da expressão de inicialização, ou quando a variável receber um valor depois e seu valor inicial não for importante.

```
i := 100 // um int  
var boiling float64 = 100 // um float64  
  
var names []string  
var err error  
var p Point
```

Como ocorre com as declarações **var**, diversas variáveis podem ser declaradas e inicializadas na mesma declaração curta de variável:

```
i, j := 0, 1
```

mas declarações com várias expressões de inicialização devem ser usadas somente quando elas ajudam a deixar o código mais legível, por exemplo, para agrupamentos pequenos e naturais como a inicialização de um loop **for**.

Tenha em mente que **:=** é uma declaração, enquanto **=** é uma atribuição. Uma declaração de diversas variáveis não deve ser confundida com uma *atribuição de tupla* (seção 2.4.1), em que cada variável do lado esquerdo

recebe o valor correspondente do lado direito:

```
i, j = j, i // troca os valores (faz um swap) de i e j
```

Assim como declarações **var** comuns, declarações curtas de variáveis podem ser usadas para chamadas a funções como **os.Open**, que devolvam dois ou mais valores:

```
f, err := os.Open(name)
if err != nil {
    return err
}
// ...usa f...
f.Close()
```

Uma questão sutil, porém importante, é que uma declaração curta de variável não *declara*, necessariamente, todas as variáveis do lado esquerdo. Se algumas delas já estiverem declaradas no *mesmo* bloco léxico (seção 2.7), a declaração curta de variável atuará como uma *atribuição* a essas variáveis.

No código a seguir, a primeira instrução declara tanto **in** quanto **err**. A segunda declara **out**, mas apenas atribui um valor à variável **err** existente.

```
in, err := os.Open(infile)
// ...
out, err := os.Create(outfile)
```

Contudo, uma declaração curta de variável deve declarar pelo menos uma nova variável, portanto o código a seguir não compilará:

```
f, err := os.Open(infile)
// ...
f, err := os.Create(outfile) // erro de compilação: não há novas variáveis
```

A correção consiste em usar uma atribuição normal na segunda instrução.

Uma declaração curta de variável atua como uma atribuição somente para as variáveis que já foram declaradas no mesmo bloco léxico; declarações em um bloco mais externo são ignoradas. Veremos exemplos disso no final do capítulo.

2.3.2 Ponteiros

Uma *variável* é um item de armazenagem que contém um valor. Variáveis criadas por declarações são identificadas por um nome, por exemplo, `x`, mas muitas variáveis são identificadas somente por expressões como `x[i]` ou `x.f`. Todas essas expressões leem o valor de uma variável, exceto quando aparecem do lado esquerdo de uma atribuição, caso em que um novo valor é atribuído à variável.

Um *ponteiro* é o *endereço* de uma variável. Desse modo, um ponteiro é o local em que um valor é armazenado. Nem todo valor tem um endereço, mas toda variável tem. Com um ponteiro, podemos ler ou atualizar o valor de uma variável *indiretamente*, sem usar ou sequer saber o nome da variável, se é que ela tem um nome.

Se uma variável for declarada como `var x int`, a expressão `&x` (“endereço de `x`”) fornece um ponteiro para uma variável inteira, isto é, um valor do tipo `*int`, lido como “ponteiro para `int`”. Se esse valor se chamar `p`, dizemos que “`p` aponta para `x`” ou, de modo equivalente, “`p` contém o endereço de `x`”. A variável à qual `p` aponta é escrita como `*p`. A expressão `*p` fornece o valor dessa variável, que é um `int`, mas como `*p` representa uma variável, ela também pode aparecer do lado esquerdo de uma atribuição, caso em que a atribuição atualiza a variável.

```
x := 1
p := &x          // p, do tipo *int, aponta para x
fmt.Println(*p)  // "1"
*p = 2           // equivalente a x = 2
fmt.Println(x)   // "2"
```

Cada componente de uma variável do tipo agregado – um campo de uma estrutura ou um elemento de um array – também é uma variável e, sendo assim, também tem um endereço.

Variáveis podem ser descritas como valores *endereçáveis*. Expressões que representam variáveis são as únicas expressões para as quais o operador `&`, isto é, *endereço-de*, é aplicável.

O valor zero de um ponteiro de qualquer tipo é `nil`. O teste `p != nil` é verdadeiro se `p` aponta para uma variável. Ponteiros são comparáveis; dois ponteiros serão iguais se e somente se apontarem para a mesma

variável ou ambos forem `nil`.

```
var x, y int
fmt.Println(&x == &x, &x == &y, &x == nil) // "true false false"
```

É perfeitamente seguro para uma função devolver o endereço de uma variável local. Por exemplo, no código a seguir, a variável local `v`, criada por esta chamada em particular a `f`, existirá mesmo depois de a chamada ter retornado, e o ponteiro `p` continuará fazendo referência a ela:

```
var p = f()
func f() *int {
    v := 1
    return &v
}
```

Cada chamada a `f` devolve um valor distinto:

```
fmt.Println(f() == f()) // "false"
```

Como um ponteiro contém o endereço de uma variável, passar um argumento do tipo ponteiro a uma função possibilita que a função atualize a variável que foi passada indiretamente. Por exemplo, a função a seguir incrementa a variável apontada pelo seu argumento e devolve o novo valor da variável de modo que ele pode ser usado em uma expressão:

```
func incr(p *int) int {
    *p++ // incrementa o valor para o qual p aponta; não altera p
    return *p
}

v := 1
incr(&v)           // efeito colateral: v agora é 2
fmt.Println(incr(&v)) // "3" (e v é 3)
```

Sempre que usamos o endereço de uma variável ou copiamos um ponteiro criamos novos *alias* (apelidos), ou maneiras de identificar a mesma variável. Por exemplo, `*p` é um alias para `v`. Aliasing (apelidamento) com ponteiros é conveniente porque nos permite acessar uma variável sem utilizar seu nome, mas é uma faca de dois gumes: para encontrar todas as instruções que acessam uma variável precisamos conhecer todos os seus aliases. Não são só os ponteiros que criam aliases: o aliasing também

ocorre quando copiamos valores de outros tipos de referência, como fatias, mapas e canais, e até mesmo estruturas, arrays e interfaces que contêm esses tipos.

Ponteiros são fundamentais para o pacote **flag**, que usa os argumentos de linha de comando de um programa para definir os valores de determinadas variáveis. Para ilustrar, a variação a seguir do comando **echo** anterior aceita duas flags opcionais: **-n** faz **echo** omitir a quebra de linha final que normalmente seria exibida e **-s sep** faz o comando separar os argumentos de saída, usando o conteúdo da string **sep** em vez de utilizar o caractere de espaço default. Como esta é a nossa quarta versão, o pacote se chama **gopl.io/ch2/echo4**.

gopl.io/ch2/echo4

```
// Echo4 exhibe seus argumentos de linha de comando.
package main

import (
    "flag"
    "fmt"
    "strings"
)

var n = flag.Bool("n", false, "omit trailing newline")
var sep = flag.String("s", " ", "separator")

func main() {
    flag.Parse()
    fmt.Print(strings.Join(flag.Args(), *sep))
    if !*n {
        fmt.Println()
    }
}
```

A função **flag.Bool** cria uma nova variável flag do tipo **bool**. Ela aceita três argumentos: o nome da flag ("**n**"), o valor default da variável (**false**) e uma mensagem que será exibida se o usuário fornecer um argumento inválido, uma flag inválida ou **-h** ou **-help**. De modo semelhante, **flag.String** aceita um nome, um valor default e uma mensagem, e cria uma variável do tipo **string**. As variáveis **sep** e **n** são ponteiros para as variáveis de flag, que devem ser acessadas indiretamente como ***sep** e ***n**.

Quando o programa é executado, ele deve chamar `flag.Parse` antes de as flags serem usadas para atualizar as variáveis de flag em relação a seus valores default. Os argumentos que não são flags estão disponíveis em `flag.Args()` como uma fatia de strings. Se `flag.Parse` encontrar um erro, ela exibe uma mensagem de uso e chama `os.Exit(2)` para terminar o programa.

Vamos executar alguns casos de teste com `echo`:

```
$ go build gopl.io/ch2/echo4
$ ./echo4 a bc def
a bc def
$ ./echo4 -s / a bc def
a/bc/def
$ ./echo4 -n a bc def
a bc def$
$ ./echo4 -help
Usage of ./echo4:
  -n      omit trailing newline
  -s string
          separator (default " ")
```

2.3.3 A função `new`

Outra maneira de criar uma variável é com a função embutida `new`. A expressão `new(T)` cria uma *variável sem nome* do tipo `T`, inicializa-a com o valor zero de `T` e devolve seu endereço, que é um valor do tipo `*T`.

```
p := new(int)    // p, do tipo *int, aponta para uma variável int sem nome
fmt.Println(*p)  // "0"
*p = 2           // define o int sem nome com 2
fmt.Println(*p)  // "2"
```

Uma variável criada com `new` não é diferente de uma variável local comum cujo endereço é acessado, exceto pelo fato de não haver necessidade de inventar (nem de declarar) um nome descartável (dummy), e podemos usar `new(T)` em uma expressão. Desse modo, `new` é apenas uma conveniência sintática, e não uma noção fundamental: as duas funções `newInt` a seguir têm comportamentos idênticos.

```
func newInt() *int {      func newInt() *int {
```

```

        return new(int)
    }
    var dummy int
    return &dummy
}

```

Cada chamada a **new** devolve uma variável distinta com um endereço único:

```

p := new(int)
q := new(int)
fmt.Println(p == q) // "false"

```

Há uma exceção a essa regra: duas variáveis cujos tipos não portem nenhuma informação e, desse modo, tenham tamanho zero, como **struct{}** ou **[0]int**, dependendo da implementação, podem ter o mesmo endereço.

O uso da função **new** é relativamente raro porque as variáveis sem nome mais comuns são do tipo estrutura, para as quais a sintaxe literal de estrutura (seção 4.4.1) é mais flexível.

Como **new** é uma função pré-declarada, e não uma palavra reservada, é possível redefinir seu nome para representar algo diferente em uma função, por exemplo:

```

func delta(old, new int) int { return new - old }

```

É claro que, em **delta**, a função embutida **new** não está disponível.

2.3.4 Tempo de vida das variáveis

O *tempo de vida* de uma variável é o intervalo de tempo durante o qual ela existe enquanto o programa executa. O tempo de vida de uma variável de nível de pacote é toda a execução de um programa. Em comparação, as variáveis locais têm tempos de vida dinâmicos: uma nova instância é criada sempre que a instrução de declaração é executada, e a variável vive até se tornar *inacessível*, momento em que sua área de armazenagem pode ser reciclada. Parâmetros e resultados de função também são variáveis locais; eles são criados sempre que a função que os engloba é chamada.

Por exemplo, no trecho de código a seguir do programa Lissajous da seção 1.4:

```

for t := 0.0; t < cycles*2*math.Pi; t += res {
    x := math.Sin(t)
    y := math.Sin(t*freq + phase)
    img.SetColorIndex(size+int(x*size+0.5), size+int(y*size+0.5), blackIndex)
}

```

a variável **t** é criada sempre que o loop **for** inicia, e novas variáveis **x** e **y** são criadas a cada iteração do loop.

Como o coletor de lixo sabe que a área de armazenagem de uma variável pode ser reciclada? A história completa é muito mais detalhada do que precisamos saber aqui, mas a ideia básica é que toda variável de nível de pacote e toda variável local de cada função ativa no momento podem, possivelmente, ser o início ou a raiz de um caminho (path) para a variável em questão, seguindo ponteiros e outros tipos de referência que, em última instância, levam à variável. Se esse caminho não existir, é sinal de que a variável se tornou inacessível, portanto não poderá mais afetar o restante do processamento.

Como o tempo de vida de uma variável é determinado somente pelo fato de ela ser ou não acessível, uma variável local pode viver mais que uma única iteração do loop que a engloba. Ela pode continuar existindo mesmo após a função que a engloba ter retornado.

Um compilador pode optar por alocar variáveis locais na heap ou na pilha, mas essa escolha, talvez surpreendentemente, não é determinada pelo fato de **var** ou **new** ter sido usada para declarar a variável.

```

var global *int

func f() {
    var x int
    x = 1
    global = &x
}

func g() {
    y := new(int)
    *y = 1
}

```

Nesse caso, **x** deve ser alocada na heap porque continua acessível a partir da variável **global** depois que **f** retornar, apesar de ser declarada como uma variável local; dizemos que **x** *escapa de f*. Por outro lado, quando **g** retorna, a variável ***y** torna-se inacessível e pode ser reciclada. Como ***y** não escapa de **g**, é seguro para o compilador alocar ***y** na pilha, apesar de

ter sido alocada com **new**. Qualquer que seja o caso, a noção de escapar não é algo com que você precise se preocupar para escrever um código correto, embora seja bom ter isso em mente durante a otimização para melhoria de desempenho, pois cada variável que escapa exige uma alocação extra de memória.

A coleta de lixo auxilia bastante na escrita de programas corretos, mas não isenta você da responsabilidade de pensar na memória. Você não precisa, explicitamente, alocar e liberar memória, mas, para escrever programas eficientes, é necessário estar ciente do tempo de vida das variáveis. Por exemplo, manter ponteiros desnecessários para objetos de vida curta em objetos de vida longa, especialmente variáveis globais, impedirá que o coletor de lixo recicle os objetos de vida curta.

2.4 Atribuições

O valor mantido por uma variável é atualizado por uma instrução de atribuição que, em sua forma mais simples, tem uma variável à esquerda do sinal = e uma expressão à direita.

```
x = 1                // variável nomeada
*p = true            // variável indireta
person.name = "bob"  // campo de estrutura
count[x] = count[x] * scale // elemento de array, de fatia ou de mapa
```

Cada um dos operadores aritméticos e binários bit a bit (bitwise) tem um *operador de atribuição* correspondente, que permite, por exemplo, reescrever a última instrução como:

```
count[x] *= scale
```

Com isso, não precisamos repetir (nem reavaliar) a expressão para a variável.

Variáveis numéricas também podem ser incrementadas e decrementadas com instruções ++ e --:

```
v := 1
v++  // é o mesmo que v = v + 1; v torna-se 2
v--  // é o mesmo que v = v - 1; v torna-se 1 novamente
```


2.4.1 Atribuição de tupla

Outra forma de atribuição, conhecida como *atribuição de tupla*, permite que diversas variáveis recebam valores de uma só vez. Todas as expressões do lado direito são avaliadas antes de qualquer variável ser atualizada, tornando essa forma mais útil quando algumas das variáveis aparecem em ambos os lados da atribuição, como acontece, por exemplo, quando trocamos os valores (fazemos swap) de duas variáveis:

```
x, y = y, x
a[i], a[j] = a[j], a[i]
```

ou quando calculamos o GCD (Greatest Common Divisor, ou Máximo Divisor Comum) de dois inteiros:

```
func gcd(x, y int) int {
    for y != 0 {
        x, y = y, x%y
    }
    return x
}
```

ou quando calculamos o enésimo número de Fibonacci iterativamente:

```
func fib(n int) int {
    x, y := 0, 1
    for i := 0; i < n; i++ {
        x, y = y, x+y
    }
    return x
}
```

A atribuição de tupla também pode deixar uma sequência de atribuições triviais mais compacta:

```
i, j, k = 2, 3, 5
```

no entanto, por questões de estilo, você deve evitar a forma de tupla se as expressões forem complexas, pois uma sequência de instruções separadas é mais fácil de ler.

Determinadas expressões, como uma chamada a uma função com vários resultados, geram diversos valores. Quando uma chamada desse tipo é usada em uma instrução de atribuição, o lado esquerdo deve ter tantas

variáveis quantos forem os resultados da função.

```
f, err = os.Open("foo.txt") // a chamada da função retorna dois valores
```

Com frequência, funções usam esses resultados adicionais para indicar algum tipo de erro, seja devolvendo um **error**, como na chamada a **os.Open**, ou um **bool**, normalmente chamado **ok**. Como veremos em capítulos mais adiante, há três operadores que, às vezes, se comportam dessa maneira também. Se uma busca em um mapa (seção 4.3), uma asserção de tipo (seção 7.10) ou uma recepção de canal (seção 8.4.2) aparecer em uma instrução em que dois resultados são esperados, cada um deles produzirá um resultado booleano adicional:

```
v, ok = m[key]      // busca em mapa  
v, ok = x.(T)       // asserção de tipo  
v, ok = <-ch        // recepção de canal
```

Como em declarações de variáveis, podemos atribuir valores que não queremos ao identificador vazio:

```
_, err = io.Copy(dst, src) // descarta o número de bytes  
_, ok = x.(T)              // verifica o tipo, mas descarta o resultado
```

2.4.2 Possibilidade de atribuição

Instruções de atribuição são uma forma explícita de atribuição, mas há muitos lugares em um programa em que uma atribuição ocorre *implicitamente*: uma chamada de função atribui implicitamente os valores dos argumentos às variáveis de parâmetro correspondentes; uma instrução **return** atribui implicitamente os operandos de **return** às variáveis de resultado correspondentes. Uma expressão literal para um tipo composto (seção 4.2) como esta fatia:

```
medals := []string{"gold", "silver", "bronze"}
```

atribui implicitamente cada elemento, como se tivessem sido escritos assim:

```
medals[0] = "gold"  
medals[1] = "silver"  
medals[2] = "bronze"
```

Os elementos de mapas e canais, embora não sejam variáveis comuns,

também estão sujeitos a atribuições implícitas semelhantes.

Uma atribuição, seja explícita ou implícita, é sempre permitida se o lado esquerdo (a variável) e o lado direito (o valor) forem do mesmo tipo. De modo geral, a atribuição é permitida somente se o valor *puder ser atribuído* ao tipo da variável.

A regra de *possibilidade de atribuição* (assignability) tem casos para diversos tipos, portanto explicaremos os casos relevantes à medida que apresentarmos cada tipo novo. Para os tipos discutidos até agora, as regras são simples: os tipos devem corresponder exatamente, e `nil` pode ser atribuído a qualquer variável do tipo interface ou referência. Constantes (seção 3.6) têm regras mais flexíveis para atribuição que evitam a necessidade de muitas conversões explícitas.

O fato de dois valores poderem ser comparados com `==` e `!=` está relacionado à possibilidade de atribuição: em qualquer comparação, deve ser possível atribuir o primeiro operando ao tipo do segundo operando, ou vice-versa. Assim como para a possibilidade de atribuição, explicaremos os casos relevantes para *comparabilidade* (comparability) quando apresentarmos cada tipo novo.

2.5 Declarações de tipos

O tipo de uma variável ou expressão define as características dos valores que ela pode assumir, por exemplo, seu tamanho (número de bits ou de elementos, por exemplo), como elas são representadas internamente, as operações intrínsecas que podem ser realizadas e os métodos associados a elas.

Em qualquer programa há variáveis que compartilham a mesma representação, mas exprimem conceitos bem diferentes. Por exemplo, um `int` pode ser usado para representar um índice de loop, um timestamp, um descritor de arquivo ou um mês; um `float64` pode representar uma velocidade em metros por segundo ou uma temperatura em uma de várias escalas, e uma `string` pode representar uma senha ou o nome de uma cor.

Uma declaração **type** define um novo *tipo nomeado* (named type) que tem o mesmo *tipo subjacente* de um tipo existente. O tipo nomeado oferece uma maneira de distinguir usos diferentes, e talvez incompatíveis, do tipo subjacente, para que eles não sejam confundidos involuntariamente.

type *nome tipo-subjacente*

As declarações de tipo aparecem com mais frequência no nível de pacote, em que o tipo nomeado é visível em todo o pacote e, se for exportado (começa com uma letra maiúscula), o nome será acessível também a outros pacotes.

Para demonstrar as declarações de tipo, vamos transformar as diferentes escalas de temperatura em tipos diferentes:

gopl.io/ch2/tempconv0

```
// Pacote tempconv realiza cálculos com temperaturas em Celsius e em Fahrenheit
package tempconv

import "fmt"

type Celsius float64
type Fahrenheit float64

const (
    AbsoluteZeroC Celsius = -273.15
    FreezingC      Celsius = 0
    BoilingC       Celsius = 100
)

func CToF(c Celsius) Fahrenheit { return Fahrenheit(c*9/5 + 32) }
func FToC(f Fahrenheit) Celsius { return Celsius((f - 32) * 5 / 9) }
```

Esse pacote define dois tipos, **Celsius** e **Fahrenheit**, para as duas unidades de temperatura. Apesar de ambos terem o mesmo tipo subjacente, **float64**, eles não são do mesmo tipo, portanto não podem ser comparados nem combinados em expressões aritméticas. Fazer a distinção entre os tipos possibilita evitar erros como combinar temperaturas inadvertidamente em duas escalas diferentes; uma *conversão* explícita de tipo, como **Celsius(t)** ou **Fahrenheit(t)**, é necessária para converter um **float64**. **Celsius(t)** e **Fahrenheit(t)** não são chamadas de função, mas conversões. Elas não alteram o valor nem a representação

de forma alguma, mas deixam a mudança de significado explícita. Por outro lado, as funções `CToF` e `FToC` fazem a conversão entre as duas escalas; elas devolvem valores diferentes.

Para todo tipo `T` há uma operação de conversão `T(x)` correspondente, que converte o valor `x` para o tipo `T`. Uma conversão de um tipo para outro é permitida se ambos tiverem o mesmo tipo subjacente ou se ambos forem tipos ponteiro sem nomes que apontam para variáveis do mesmo tipo subjacente; essas conversões alteram o tipo, mas não mudam a representação do valor. Se `x` puder ser atribuído a `T`, uma conversão é permitida, mas normalmente será redundante.

Conversões também são permitidas entre tipos numéricos, e entre string e alguns tipos de fatias, como veremos no próximo capítulo. Essas conversões podem mudar a representação do valor. Por exemplo, converter um número de ponto flutuante para um inteiro faz qualquer parte fracionária ser descartada, e converter uma string para uma fatia `[]byte` aloca uma cópia dos dados da string. Qualquer que seja o caso, uma conversão jamais falha em tempo de execução.

O tipo subjacente de um tipo nomeado determina sua estrutura e sua representação, além de definir o conjunto de operações intrínsecas que ele aceita, ou seja, o efeito é o mesmo se o tipo subjacente tivesse sido usado diretamente. Isso quer dizer que os operadores aritméticos funcionam para `Celsius` e `Fahrenheit` do mesmo modo como funcionam para `float64`, como esperado.

```
fmt.Printf("%g\n", BoilingC-FreezingC) // "100" °C
boilingF := CToF(BoilingC)
fmt.Printf("%g\n", boilingF-CToF(FreezingC)) // "180" °F
fmt.Printf("%g\n", boilingF-FreezingC) // erro de compilação:
// incompatibilidade de tipos
```

Os operadores de comparação como `==` e `<` também podem ser usados para comparar um valor de um tipo nomeado com outro valor do mesmo tipo, ou com um valor de um tipo anônimo com o mesmo tipo subjacente. Porém, dois valores de tipos nomeados diferentes não podem ser diretamente comparados:

```

var c Celsius
var f Fahrenheit
fmt.Println(c == 0) // "true"
fmt.Println(f >= 0) // "true"
fmt.Println(c == f) // erro de compilação: incompatibilidade de tipos
fmt.Println(c == Celsius(f)) // "true"!

```

Observe o último caso com atenção. Apesar de seu nome, a conversão de tipo **Celsius(f)** não altera o valor de seu argumento, apenas o seu tipo. O teste é verdadeiro porque **c** e **f** são iguais a zero.

Um tipo nomeado pode ser conveniente quanto à notação se ele ajuda a evitar a necessidade de escrever os tipos complexos repetidamente. Há pouca vantagem quando o tipo subjacente é simples como **float64**, mas a vantagem é grande para tipos complexos, como veremos quando discutirmos as estruturas.

Tipos nomeados também possibilitam definir novos comportamentos para os valores desse tipo. Esses comportamentos são expressos na forma de um conjunto de funções associadas ao tipo, chamadas de *métodos* do tipo. Daremos uma olhada nos métodos com detalhes no capítulo 6, mas daremos uma amostra do mecanismo de funcionamento aqui.

A declaração a seguir, em que o parâmetro **c** do tipo **Celsius** aparece antes do nome da função, associa ao tipo **Celsius** um método chamado **String**, que devolve o valor numérico de **c** seguido de °C:

```

func (c Celsius) String() string { return fmt.Sprintf("%g°C", c) }

```

Muitos tipos declaram um método **String** dessa forma porque ele controla como os valores desse tipo aparecem quando exibidos na forma de string pelo pacote **fmt**, como veremos na seção 7.1.

```

c := FToC(212.0)
fmt.Println(c.String()) // "100°C"
fmt.Printf("%v\n", c)   // "100°C"; não há necessidade de chamar String
                        // explicitamente
fmt.Printf("%s\n", c)   // "100°C"
fmt.Println(c)          // "100°C"
fmt.Printf("%g\n", c)   // "100"; não chama String
fmt.Println(float64(c)) // "100"; não chama String

```

2.6 Pacotes e arquivos

Pacotes em Go têm o mesmo propósito das bibliotecas ou módulos em outras linguagens; eles suportam modularidade, encapsulamento, compilação separada e reutilização. O código-fonte de um pacote está em um ou mais arquivos `.go`, normalmente em um diretório cujo nome termina com o path de importação: por exemplo, os arquivos do pacote `gopl.io/ch1/helloworld` são armazenados no diretório `$GOPATH/src/gopl.io/ch1/helloworld`.

Cada pacote serve como um *name space* (espaço de nomes) separado para suas declarações. No pacote `image`, por exemplo, o identificador `Decode` refere-se a uma função diferente do identificador de mesmo nome no pacote `unicode/utf16`. Para referenciar uma função de fora de seu pacote, devemos *qualificar* o identificador para deixar explícito se queremos dizer `image.Decode` ou `utf16.Decode`.

Pacotes também nos permitem ocultar informações, controlando quais nomes são visíveis fora do pacote, ou seja, são *exportados*. Em Go, uma regra simples determina quais identificadores são exportados e quais não são: identificadores exportados começam com uma letra maiúscula.

Para mostrar o básico, suponha que nosso software de conversão de temperatura tenha se tornado popular e que queremos disponibilizá-lo à comunidade Go na forma de um novo pacote. Como podemos fazer isso?

Vamos criar um pacote chamado `gopl.io/ch2/tempconv`, que é uma variação do exemplo anterior. (Neste caso, criamos uma exceção à nossa regra usual de numerar os exemplos em sequência para que o path do pacote possa ser mais realista.) O pacote propriamente dito é armazenado em dois arquivos para mostrar como as declarações em arquivos separados de um pacote são acessadas; na vida real, um pacote minúsculo como esse precisaria de apenas um arquivo.

Colocamos as declarações de tipos, as constantes e os métodos em `tempconv.go`:

`gopl.io/ch2/tempconv`

```
// Pacote tempconv realiza conversões de Celsius e Fahrenheit.
package tempconv

import "fmt"

type Celsius float64
type Fahrenheit float64
const (
    AbsoluteZeroC Celsius = -273.15
    FreezingC      Celsius = 0
    BoilingC       Celsius = 100
)
func (c Celsius) String() string { return fmt.Sprintf("%g°C", c) }
func (f Fahrenheit) String() string { return fmt.Sprintf("%g°F", f) }
```

As funções de conversão estão em **conv.go**:

```
package tempconv

// CToF converte uma temperatura em Celsius para Fahrenheit.
func CToF(c Celsius) Fahrenheit { return Fahrenheit(c*9/5 + 32) }

// FToC converte uma temperatura em Fahrenheit para Celsius.
func FToC(f Fahrenheit) Celsius { return Celsius((f - 32) * 5 / 9) }
```

Cada arquivo começa com uma declaração **package** que define o nome do pacote. Quando o pacote é importado, seus membros são referenciados como **tempconv.CToF** e assim por diante. Nomes no nível de pacote, como os tipos e as constantes declarados em um arquivo de um pacote, são visíveis a todos os demais arquivos do pacote, como se todo o código-fonte estivesse em um único arquivo. Observe que **tempconv.go** importa **fmt**, mas **conv.go** não o faz, pois ele não usa nada que está em **fmt**.

Como os nomes **const** do nível de pacote começam com letras maiúsculas, eles também são acessíveis por meio de nomes qualificados como **tempconv.AbsoluteZeroC**:

```
fmt.Printf("Brrrr! %v\n", tempconv.AbsoluteZeroC) // "Brrrr! -273.15°C"
```

Para converter uma temperatura em Celsius para Fahrenheit em um pacote que importa **gopl.io/ch2/tempconv**, podemos escrever o código a seguir:

```
fmt.Println(tempconv.CToF(tempconv.BoilingC)) // "212°F"
```

O *comentário doc* (doc comment; seção 10.7.4), que está imediatamente

antes da declaração **package**, documenta o pacote como um todo. Por convenção, ele deve começar com uma frase resumida no estilo ilustrado. Somente um arquivo em cada pacote deve ter um comentário doc de pacote. Comentários doc extensos geralmente são colocados em um arquivo próprio, chamado **doc.go** por convenção.

Exercício 2.1: Acrescente tipos, constantes e funções em **tempconv** para processar temperaturas na escala Kelvin, em que zero Kelvin corresponde a $-273,15\text{ }^{\circ}\text{C}$ e uma diferença de 1 K tem a mesma magnitude de $1\text{ }^{\circ}\text{C}$.

2.6.1 Importações

Em um programa Go, todo pacote é identificado por uma string única chamada *path de importação* (import path). São as strings que aparecem em uma declaração **import**, por exemplo, **"gopl.io/ch2/tempconv"**. A especificação da linguagem não define a origem dessas strings nem o que elas significam; interpretá-las é responsabilidade das ferramentas. Ao usar a ferramenta **go** (Capítulo 10), um path de importação representa um diretório contendo um ou mais arquivos-fonte Go que, juntos, compõem o pacote.

Além de seu path de importação, cada pacote tem um *nome de pacote* (package name), que é o nome curto (e não necessariamente único) que aparece em sua declaração **package**. Por convenção, o nome de um pacote coincide com o último segmento de seu path de importação, facilitando prever que o nome do pacote de **gopl.io/ch2/tempconv** é **tempconv**.

Para usar **gopl.io/ch2/tempconv**, devemos importá-lo:

gopl.io/ch2/cf

```
// Cf converte seu argumento numérico para Celsius e Fahrenheit
package main

import (
    "fmt"
    "os"
    "strconv"
    "gopl.io/ch2/tempconv"
)
```

```

func main() {
    for _, arg := range os.Args[1:] {
        t, err := strconv.ParseFloat(arg, 64)
        if err != nil {
            fmt.Fprintf(os.Stderr, "cf: %v\n", err)
            os.Exit(1)
        }
        f := tempconv.Fahrenheit(t)
        c := tempconv.Celsius(t)
        fmt.Printf("%s = %s, %s = %s\n",
            f, tempconv.FToC(f), c, tempconv.CToF(c))
    }
}

```

A declaração de importação vincula um nome curto ao pacote importado, e pode ser usado para referir-se ao conteúdo do pacote em todo o arquivo. A declaração `import` anterior nos permite referenciar nomes em `gopl.io/ch2/tempconv` usando um *identificador qualificado*, por exemplo, `tempconv.CToF`. Por padrão, o nome curto é o nome do pacote — `tempconv` nesse caso —, mas uma declaração de importação pode especificar um nome alternativo para evitar um conflito (seção 10.4).

O programa `cf` converte um único argumento numérico de linha de comando para seu valor tanto em Celsius quanto em Fahrenheit:

```

$ go build gopl.io/ch2/cf
$ ./cf 32
32°F = 0°C, 32°C = 89.6°F
$ ./cf 212
212°F = 100°C, 212°C = 413.6°F
$ ./cf -40
-40°F = -40°C, -40°C = -40°F

```

É um erro importar um pacote e não o referenciar. Essa verificação ajuda a eliminar dependências que se tornam desnecessárias à medida que o código evolui, embora possa ser irritante durante a depuração, pois comentar uma linha de código como `log.Print("got here!")` pode remover a única referência ao pacote chamado `log`, fazendo o compilador gerar um erro. Nessa situação, você deve comentar ou apagar o `import` desnecessário.

Melhor ainda, use a ferramenta golang.org/x/tools/cmd/goimports, que insere e remove pacotes automaticamente da declaração de importação conforme for necessário; a maioria dos editores pode ser configurada para executar **goimports** sempre que você salvar um arquivo. Assim como a ferramenta **gofmt**, ela também faz um pretty-print dos arquivos-fonte Go no formato canônico.

Exercício 2.2: Escreva um programa de conversão de unidades de propósito geral, análogo ao **cf**, que leia números de seus argumentos de linha de comando ou da entrada-padrão se não houver argumentos, e converta cada número em unidades como temperatura em Celsius e em Fahrenheit, comprimento em pés e metros, peso em libras e quilogramas e operações semelhantes.

2.6.2 Inicialização de pacotes

A inicialização de pacotes começa pela inicialização de variáveis de nível de pacote na ordem em que elas são declaradas, exceto que as dependências são resolvidas antes:

```
var a = b + c // a é inicializada em terceiro lugar, com 3
var b = f()   // b é inicializada em segundo lugar, com 2, chamando f
var c = 1     // c é inicializada em primeiro lugar, com 1
func f() int { return c + 1 }
```

Se o pacote tiver vários arquivos **.go**, eles serão inicializados na ordem em que os arquivos são fornecidos ao compilador; a ferramenta **go** ordena arquivos **.go** pelo nome antes de chamar o compilador.

Cada variável declarada no nível de pacote nasce com o valor de sua expressão de inicialização, se houver, mas para algumas variáveis, como tabelas de dados, uma expressão de inicialização pode não ser a maneira mais simples de definir seu valor inicial. Nesse caso, o mecanismo da função **init** pode ser mais simples. Qualquer arquivo pode conter qualquer quantidade de funções cuja declaração seja simplesmente:

```
func init() { /* ... */ }
```

Essas funções **init** não podem ser chamadas nem referenciadas, mas em

outros aspectos são funções normais. Em cada arquivo, funções `init` são automaticamente executadas quando o programa inicia e na ordem em que elas são declaradas.

Um pacote é inicializado de cada vez, na ordem das importações no programa – em primeiro lugar as dependências – portanto um pacote `p` que importe `q` pode ter certeza de que `q` estará totalmente inicializado antes que a inicialização de `p` comece. A inicialização ocorre de baixo para cima; o pacote `main` é o último a ser inicializado. Dessa forma, todos os pacotes são completamente inicializados antes que a função `main` da aplicação comece.

O próximo pacote define uma função `PopCount` que devolve o número de bits setados, ou seja, bits cujo valor é 1, em um valor `uint64`, que é a sua *população* (population count). Uma função `init` é usada para pré-calcular uma tabela de resultados, `pc`, para cada valor possível de 8 bits, para que a função `PopCount` não precise executar 64 passos, mas possa simplesmente devolver a soma de oito consultas em uma tabela. (Definitivamente, esse *não* é o algoritmo mais rápido para contar bits, mas é conveniente para ilustrar as funções `init` e para mostrar como pré-calcular uma tabela de valores, o que, muitas vezes, é uma técnica de programação útil.)

gopl.io/ch2/popcount

```
package popcount

// pc[i] é a população de i
var pc [256]byte

func init() {
    for i := range pc {
        pc[i] = pc[i/2] + byte(i&1)
    }
}

// PopCount devolve a população (número de bits definidos) de x
func PopCount(x uint64) int {
    return int(pc[byte(x>>(0*8))] +
               pc[byte(x>>(1*8))] +
               pc[byte(x>>(2*8))] +
               pc[byte(x>>(3*8))] +
               pc[byte(x>>(4*8))] +
               pc[byte(x>>(5*8))] +
               pc[byte(x>>(6*8))] +
               pc[byte(x>>(7*8))])
```

```

        pc[byte(x>>(5*8))] +
        pc[byte(x>>(6*8))] +
        pc[byte(x>>(7*8))])
    }

```

Observe que o loop **range** em **init** usa somente o índice; o valor não é necessário e, por isso, não precisa ser incluído. O loop poderia também ter sido escrito assim:

```
for i, _ := range pc {
```

Veremos outros usos de funções **init** na próxima seção e na seção 10.5.

Exercício 2.3: Reescreva **PopCount** para que use um loop no lugar de uma expressão única. Compare o desempenho das duas versões. (A seção 11.4 mostra como comparar o desempenho de diferentes implementações de forma sistemática.)

Exercício 2.4: Escreva uma versão de **PopCount** que conte bits deslocando seu argumento pelas 64 posições dos bits, testando o bit mais à direita a cada vez. Compare seu desempenho com a versão que faz consultas na tabela.

Exercício 2.5: A expressão **x&(x-1)** limpa o bit diferente de zero mais à direita de **x**. Escreva uma versão de **PopCount** que conte bits usando esse fato e avalie seu desempenho.

2.7 Escopo

Uma declaração associa um nome a uma entidade do programa, por exemplo, a uma função ou uma variável. O *escopo* de uma declaração é a parte do código-fonte em que um uso do nome declarado refere-se a essa declaração.

Não confunda escopo com tempo de vida. O escopo de uma declaração é uma região do texto do programa: é uma propriedade de tempo de compilação. O tempo de vida de uma variável é o intervalo de tempo durante a execução em que a variável pode ser referenciada por outras partes do programa; é uma propriedade de tempo de execução.

Um *bloco* sintático é uma sequência de instruções entre chaves, como

aquelas em torno do corpo de uma função ou de um loop. Um nome declarado em um bloco sintático não é visível fora desse bloco. O bloco engloba suas declarações e determina seu escopo. Podemos generalizar essa noção de blocos para incluir outros agrupamentos de declarações que não estão explicitamente cercados por chaves no código-fonte; chamaremos a isso de *blocos léxicos* (lexical blocks). Há um bloco léxico para todo o código-fonte, chamado *bloco universal* (universe block), para cada pacote, para cada arquivo, para cada instrução **for**, **if** e **switch**, para cada caso em uma instrução **switch** ou **select** e, é claro, para cada bloco sintático explícito.

O bloco léxico de uma declaração determina seu escopo, que pode ser grande ou pequeno. As declarações de tipos, funções e constantes embutidos como **int**, **len** e **true** estão no bloco universal e podem ser referenciadas por todo o programa. Declarações fora de qualquer função, ou seja, no *nível de pacote*, podem ser referenciadas a partir de qualquer arquivo no mesmo pacote. Pacotes importados, como **fmt** no exemplo **tempconv**, são declarados no *nível de arquivo*, portanto podem ser referenciados a partir do mesmo arquivo, mas não de outro arquivo do mesmo pacote sem outro **import**. Muitas declarações, como a da variável **c** na função **tempconv.CToF**, são *locais*, assim podem ser referenciadas somente de dentro da mesma função ou, talvez, apenas de uma parte dela.

O escopo de um rótulo de controle de fluxo, como usado por instruções **break**, **continue** e **goto**, é toda a função que o engloba.

Um programa pode conter várias declarações de mesmo nome, desde que cada declaração esteja em um bloco léxico diferente. Por exemplo, podemos declarar uma variável local com o mesmo nome de uma variável no nível de pacote. Ou, como mostramos na seção 2.3.3, podemos declarar um parâmetro de função chamado **new**, apesar de uma função com esse nome estar pré-declarada no bloco universal. Não exagere, porém, quanto maior o escopo da nova declaração, maiores serão as chances de causar surpresas ao leitor.

Quando encontra uma referência a um nome, o compilador procura uma declaração, começando pelo bloco léxico mais interno que a engloba, trabalhando em direção ao bloco universal. Se não encontrar uma declaração, o compilador informará um erro de “nome não declarado”. Se um nome estiver declarado tanto em um bloco mais externo quanto em um bloco mais interno, a declaração interna será encontrada antes. Nesse caso, dizemos que a declaração interna *encobre* ou *oculta* a declaração externa, deixando-a inacessível:

```
func f() {}  
var g = "g"  
func main() {  
    f := "f"  
    fmt.Println(f) // "f"; variável local f encobre a função f do nível  
                  // de pacote  
    fmt.Println(g) // "g"; variável do nível de pacote  
    fmt.Println(h) // erro de compilação: h não está definido  
}
```

Em uma função, blocos léxicos podem estar aninhados em qualquer profundidade, portanto uma declaração local pode encobrir outra. A maioria dos blocos é criada por construções de controle de fluxo, como instruções **if** e loops **for**. O programa a seguir tem três variáveis diferentes de nome **x** porque cada declaração aparece em um bloco léxico distinto. (Este exemplo mostra as regras de escopo, e não um bom estilo!)

```
func main() {  
    x := "hello!"  
    for i := 0; i < len(x); i++ {  
        x := x[i]  
        if x != '!' {  
            x := x + 'A' - 'a'  
            fmt.Printf("%c", x) // "HELLO" (uma letra por iteração)  
        }  
    }  
}
```

Nas expressões `x[i]` e `x + 'A' - 'a'`, cada uma se refere a uma declaração de **x** de um bloco mais externo; exploraremos isso em breve. (Observe que a última expressão *não* é equivalente a `unicode.ToUpper`.)

Como mencionamos, nem todos os blocos léxicos correspondem a sequências de instruções explicitamente delimitadas por chaves; alguns são implícitos. O loop **for** anterior cria dois blocos léxicos: o bloco explícito para o corpo do loop e um bloco implícito que, adicionalmente, engloba as variáveis declaradas pela cláusula de inicialização, como **i**. O escopo de uma variável declarada no bloco implícito é formado pela condição, a pós-instrução (**i++**) e o corpo da instrução **for**.

O próximo exemplo também tem três variáveis de nome **x**, cada uma declarada em um bloco diferente – uma no corpo da função, uma no bloco da instrução **for** e outra no corpo do loop – mas somente dois dos blocos são explícitos:

```
func main() {  
    x := "hello"  
    for _, x := range x {  
        x := x + 'A' - 'a'  
        fmt.Printf("%c", x) // "HELLO" (uma letra por iteração)  
    }  
}
```

Assim como os loops **for**, as instruções **if** e **switch** também criam blocos implícitos, além dos blocos de seus corpos. O código da cadeia **if-else** a seguir mostra o escopo de **x** e de **y**:

```
if x := f(); x == 0 {  
    fmt.Println(x)  
} else if y := g(x); x == y {  
    fmt.Println(x, y)  
} else {  
    fmt.Println(x, y)  
}  
fmt.Println(x, y) // erro de compilação: x e y não são visíveis aqui
```

A segunda instrução **if** está aninhada na primeira, portanto variáveis declaradas no inicializador da primeira instrução são visíveis na segunda. Regras semelhantes aplicam-se a cada caso de uma instrução **switch**: há um bloco para a condição e um bloco para o corpo de cada caso.

No nível de pacote, a ordem em que as declarações aparecem não tem efeito em seus escopos, portanto uma declaração pode referir-se a si

mesma ou a outra que está depois dela, permitindo declarar tipos e funções recursivos ou mutuamente recursivos. Contudo, o compilador informará um erro se uma declaração de constante ou de variável referir-se a ela mesma.

Neste programa:

```
if f, err := os.Open(fname); err != nil {    // erro de compilação: f não
                                              // é usado
    return err
}
f.ReadByte()    // erro de compilação: f não está definido
f.Close()       // erro de compilação: f não está definido
```

o escopo de `f` é somente a instrução `if`, portanto `f` não é acessível às instruções que estão a seguir, resultando erros de compilação. Conforme o compilador, você pode obter um erro adicional que informa que a variável local `f` não foi usada.

Assim, geralmente é necessário declarar `f` antes da condição para que ela seja acessível depois dela:

```
f, err := os.Open(fname)
if err != nil {
    return err
}
f.ReadByte()
f.Close()
```

Você pode se sentir tentado a evitar declarar `f` e `err` no bloco externo movendo as chamadas a `ReadByte` e a `Close` para dentro de um bloco `else`:

```
if f, err := os.Open(fname); err != nil {
    return err
} else {
    // f e err são visíveis aqui também
    f.ReadByte()
    f.Close()
}
```

mas a prática comum em Go é lidar com o erro no bloco `if` e então retornar, para que o caminho de execução bem-sucedido não fique

indentado.

Declarações curtas de variáveis exigem consciência do escopo. Considere o próximo programa, que começa obtendo seu diretório de trabalho atual e salva-o em uma variável no nível de pacote. Isso poderia ser feito com a chamada a `os.Getwd` na função `main`, mas seria melhor separar essa tarefa da lógica principal, especialmente se a falha em obter o diretório for um erro fatal. A função `log.Fatalf` exibe uma mensagem e chama `os.Exit(1)`.

```
var cwd string

func init() {
    cwd, err := os.Getwd() // erro de compilação: cwd não é usado
    if err != nil {
        log.Fatalf("os.Getwd failed: %v", err)
    }
}
```

Como nem `cwd` nem `err` estão declaradas no bloco da função `init`, a instrução `:=` declara ambas como variáveis locais. A declaração interna de `cwd` deixa a declaração externa inacessível, portanto a instrução não atualiza a variável `cwd` do nível de pacote como se pretendia.

Compiladores atuais de Go detectam que a variável local `cwd` não é usada e informa isso como um erro, mas não se exige rigorosamente que eles façam essa verificação. Além do mais, uma mudança pequena, como o acréscimo de uma instrução de logging que se refira ao `cwd` local anularia a verificação.

```
var cwd string

func init() {
    cwd, err := os.Getwd() // NOTA: errado!
    if err != nil {
        log.Fatalf("os.Getwd failed: %v", err)
    }
    log.Printf("Working directory = %s", cwd)
}
```

A variável global `cwd` permanece não inicializada, e a saída aparentemente normal de log oculta o bug.

Há várias maneiras de lidar com esse problema em potencial. A solução mais direta é evitar `:=` declarando `err` com um `var` separado:

```
var cwd string

func init() {
    var err error
    cwd, err = os.Getwd()
    if err != nil {
        log.Fatalf("os.Getwd failed: %v", err)
    }
}
```

Já vimos como pacotes, arquivos, declarações e instruções expressam a estrutura dos programas. Nos próximos dois capítulos, daremos uma olhada na estrutura dos dados.

3

Tipos de dados básicos

No final, tudo se reduz a bits, é claro, mas os computadores trabalham basicamente com números de tamanho fixo chamados *palavras* (words), que são interpretadas como inteiros, números de ponto flutuante, conjuntos de bits ou endereços de memória e, então, são combinadas em agregados maiores que representam pacotes, pixels, portfólios, poesia e tudo o mais. A linguagem Go apresenta várias maneiras de organizar dados, com um espectro de tipos de dados que, de um lado correspondem aos recursos do hardware e, de outro, oferecem o que os programadores precisam para representar convenientemente estruturas de dados complicadas.

Os tipos de Go classificam-se em quatro grupos: *tipos básicos*, *tipos agregados*, *tipos referência* e *tipos interface*. Os tipos básicos, que são o assunto deste capítulo, incluem números, strings e booleanos. Os tipos agregados – arrays (seção 4.1) e estruturas (seção 4.4) – formam tipo de dados mais complicados pela combinação de valores de diversos tipos mais simples. Tipos referência são um grupo diversificado que inclui ponteiros (seção 2.3.2), fatias (seção 4.2), mapas (seção 4.3), funções (capítulo 5) e canais (capítulo 8), mas o que eles têm em comum é que todos se referem a variáveis ou estados do programa *indiretamente*, portanto o efeito de uma operação aplicada a uma referência é observado em todas as cópias dessa referência. Por fim, discutiremos os tipos interface no capítulo 7.

3.1 Inteiros

Os tipos numéricos de Go incluem vários tamanhos de inteiros, números de ponto flutuante e números complexos. Cada tipo numérico determina

o tipo e o fato de o valor ter sinal ou não. Vamos começar pelos inteiros.

Go permite operações aritméticas com inteiros com sinal (signed) e sem sinal (unsigned). Há quatro tamanhos distintos para inteiros com sinal – 8, 16, 32 e 64 bits – representados pelos tipos `int8`, `int16`, `int32` e `int64`, e versões sem sinal correspondentes: `uint8`, `uint16`, `uint32` e `uint64`.

Há também dois tipos simplesmente chamados de `int` e `uint` que são os tamanhos naturais e mais eficientes para inteiros com e sem sinal em uma plataforma em particular: `int` é, de longe, o tipo numérico mais usado. Esses dois tipos têm o mesmo tamanho, seja 32 ou 64 bits, mas não se deve pressupô-lo: compiladores diferentes poderão fazer escolhas diferentes, mesmo em um hardware idêntico.

O tipo `rune` é sinônimo para `int32` e, por convenção, indica que um valor é um código Unicode (Unicode code point). Os dois nomes podem ser usados indistintamente. De modo semelhante, o tipo `byte` é sinônimo de `uint8` e enfatiza que o valor é um dado bruto, e não uma quantidade numérica pequena.

Por fim, existe um tipo inteiro sem sinal `uintptr`, cujo tamanho não é especificado, mas é suficiente para armazenar todos os bits de um valor de ponteiro. O tipo `uintptr` é usado somente em programação de baixo nível, por exemplo, na fronteira entre um programa Go e uma biblioteca C ou um sistema operacional. Veremos exemplos disso quando lidarmos com o pacote `unsafe` no capítulo 13.

Independentemente de seu tamanho, `int`, `uint` e `uintptr` são tipos diferentes de seus irmãos de tamanhos explícitos. Desse modo, `int` não é do mesmo tipo que `int32`, mesmo se o tamanho natural dos inteiros for 32 bits, e é preciso fazer uma conversão explícita para usar um valor `int` nos lugares em que um `int32` é necessário, e vice-versa.

Números com sinal são representados na forma de complemento de 2, em que o bit de mais alta ordem é reservado para o sinal do número e o intervalo de valores de um número de n bits varia de -2^{n-1} a $2^{n-1}-1$. Inteiros sem sinal usam o conjunto completo de bits para valores não negativos e, desse modo, seu intervalo varia de 0 a 2^n-1 . Por exemplo, o

intervalo de `int8` varia de -128 a 127, enquanto o intervalo de `uint8` é de 0 a 255.

Os operadores binários de Go para aritmética, lógica e comparação estão listados a seguir, na ordem decrescente de precedência:

*	/	%	<<	>>	&	&^
+	-		^			
==	!=	<	<=	>	>=	
&&						

Há apenas cinco níveis de precedência para operadores binários. Operadores no mesmo nível associam-se à esquerda, portanto parênteses podem ser necessários por questões de clareza ou para que os operadores sejam avaliados na ordem desejada em uma expressão como `mask & (1 << 28)`.

Cada operador nas duas primeiras linhas da tabela anterior, por exemplo `+`, tem um *operador de atribuição* correspondente, como `+=`, que pode ser usado para abreviar uma instrução de atribuição.

Os operadores aritméticos `+`, `-`, `*` e `/` podem ser aplicados a inteiros, números de ponto flutuante e números complexos, mas o operador de resto `%` aplica-se somente a inteiros. O comportamento de `%` para números negativos varia entre linguagens de programação. Em Go, o sinal do resto é sempre igual ao sinal do dividendo, portanto `-5%3` e `-5%-3` são ambos iguais a `-2`. O comportamento de `/` depende de seus operandos serem inteiros, assim `5.0/4.0` é `1.25`, mas `5/4` é `1` porque a divisão de inteiros trunca o resultado na direção do valor zero.

Se o resultado de uma operação aritmética, seja com ou sem sinal, tiver mais bits do que possam ser representados no tipo do resultado, dizemos que há *transbordamento* (overflow). Os bits de mais alta ordem que não couberem serão silenciosamente descartados. Se o número original for um tipo com sinal, o resultado poderá ser negativo se o bit mais à esquerda for 1, como no exemplo com `int8` a seguir:

```
var u uint8 = 255
```

```
fmt.Println(u, u+1, u*u) // "255 0 1"
var i int8 = 127
fmt.Println(i, i+1, i*i) // "127 -128 1"
```

Dois inteiros de mesmo tipo podem ser comparados com os operadores binários de comparação a seguir; o tipo de uma expressão de comparação é um booleano.

`==` igual a

`!=` diferente de

`<` menor que

`<=` menor ou igual a

`>` maior que

`>=` maior ou igual a

Todos os valores de tipos básicos – booleanos, números e strings – são *comparáveis*, o que quer dizer que dois valores de mesmo tipo podem ser comparados com os operadores `==` e `!=`. Além disso, inteiros, números de ponto flutuante e strings são *ordenados* pelos operadores de comparação. Os valores de vários outros tipos não são comparáveis, e nenhum outro tipo é ordenado. À medida que conhecermos cada tipo, apresentaremos as regras que governam a *possibilidade de comparação* (comparability) entre seus valores.

Existem também operadores unários de adição e de subtração:

+ positivo unário (sem efeito)

- negação unária

Para inteiros, `+x` é a versão abreviada de `0+x`, e `-x` é a versão abreviada de `0-x`; para números de ponto flutuante e complexos, `+x` é apenas `x` e `-x` é a negação de `x`.

Go também oferece os operadores binários bit a bit (bitwise) a seguir, em que os quatro primeiros tratam seus operadores como padrões de bits, sem os conceitos aritméticos de sinal ou transporte (carry ou vai-um):

& E bit a bit

| OU bit a bit

^ XOR (OU exclusivo) bit a bit

&^ limpeza de bit (AND NOT)

<< deslocamento à esquerda (left shift)

>> deslocamento à direita (right shift)

O operador ^ é o OU exclusivo (XOR) quando usado como operador binário, mas quando usado como prefixo de um operando, ou seja, como operador unário, é a negação bit a bit ou complemento, isto é, retorna um valor em que cada bit de seu operando estará invertido. O operador &^ serve para limpeza de bits (AND NOT): na expressão $z = x \ \&\^ \ y$, cada bit de z será 0 se o bit correspondente de y for 1; caso contrário, será igual ao bit correspondente de x .

O código a seguir mostra como as operações bit a bit podem ser usadas para interpretar um valor `uint8` como um conjunto compacto e eficiente de 8 bits independentes (um bitset). O verbo `%b` de `Printf` é usado para exibir os dígitos binários de um número; `08` modifica `%b` (um advérbio!) para preencher o resultado com zeros de modo a haver exatamente 8 dígitos.

```
var x uint8 = 1<<1 | 1<<5
var y uint8 = 1<<1 | 1<<2

fmt.Printf("%08b\n", x)    // "00100010", o conjunto {1, 5}
fmt.Printf("%08b\n", y)    // "00000110", o conjunto {1, 2}

fmt.Printf("%08b\n", x&y)  // "00000010", a intersecção {1}
fmt.Printf("%08b\n", x|y)  // "00100110", a união {1, 2, 5}
fmt.Printf("%08b\n", x^y)  // "00100100", a diferença simétrica {2, 5}
fmt.Printf("%08b\n", x&^y) // "00100000", a diferença {5}

for i := uint(0); i < 8; i++ {
    if x&(1<<i) != 0 { // teste de pertinência
        fmt.Println(i) // "1", "5"
    }
}

fmt.Printf("%08b\n", x<<1) // "01000100", o conjunto {2, 6}
fmt.Printf("%08b\n", x>>1) // "00010001", o conjunto {0, 4}
```


(A seção 6.5 mostra uma implementação de conjuntos de inteiros que podem ser muito maiores que um byte.)

Nas operações de deslocamento $x \ll n$ e $x \gg n$, o operando n determina o número de posições de bits a ser deslocado e deve ser sem sinal; o operando x pode ter sinal ou não. Do ponto de vista aritmético, um deslocamento para a esquerda $x \ll n$ é equivalente à multiplicação por 2^n e um deslocamento para a direita $x \gg n$ é equivalente à divisão inteira por 2^n .

Deslocamentos à esquerda preenchem os bits vagos com zeros, assim como os deslocamentos à direita de números sem sinal, porém deslocamentos à direita de números com sinal preenchem os bits vagos com cópias do bit de sinal. Por esse motivo, é importante usar a aritmética sem sinal quando você estiver tratando um inteiro como um padrão de bits.

Embora Go ofereça números e operações aritméticas sem sinal, temos a tendência de usar a forma `int` com sinal, mesmo para quantidades que não podem ser negativas, por exemplo, o tamanho de um array, apesar de `uint` parecer uma escolha mais óbvia. De fato, a função embutida `len` devolve um `int` com sinal, como no loop a seguir, que anuncia as medalhas em ordem inversa:

```
medals := []string{"gold", "silver", "bronze"}
for i := len(medals) - 1; i >= 0; i-- {
    fmt.Println(medals[i])    // "bronze", "silver", "gold"
}
```

A alternativa seria desastrosa. Se `len` devolvesse um número sem sinal, `i` também seria um `uint`, e a condição `i >= 0` sempre seria verdadeira por definição. Após a terceira iteração, em que `i == 0`, a instrução `i--` faria `i` tornar-se o valor máximo de `uint` (por exemplo, $2^{64}-1$), e não -1 , e a avaliação de `medals[i]` falharia em tempo de execução – ocorreria um *pânico* (seção 5.9) – por causa da tentativa de acessar um elemento fora dos limites da fatia.

Por esse motivo, números sem sinal tendem a ser usados somente quando

seus operadores bit a bit ou operadores aritméticos peculiares forem necessários, por exemplo, quando implementamos conjuntos de bits (bitsets), fazemos parse de formatos de arquivos binários ou para hashing e criptografia. Normalmente, eles não são usados em casos que sejam apenas de quantidades não negativas.

Em geral, uma conversão explícita é necessária para converter um valor de um tipo para outro, e operadores binários para aritmética e lógica (exceto os deslocamentos) devem ter operandos do mesmo tipo. Embora isso, ocasionalmente, resulte em expressões mais longas, também elimina toda uma classe de problemas e deixa os programas mais fáceis de entender.

Como um exemplo familiar em outros contextos, considere esta sequência:

```
var apples int32 = 1
var oranges int16 = 2
var compote int = apples + oranges    // erro de compilação
```

Tentar compilar essas três declarações gera uma mensagem de erro:

```
invalid operation: apples + oranges (mismatched types int32 and int16)
```

Essa incompatibilidade de tipos pode ser corrigida de várias maneiras, em que a mais simples é converter tudo para um tipo comum:

```
var compote = int(apples) + int(oranges)
```

Conforme descrito na seção 2.5, para todo tipo T , a operação de conversão $T(x)$ converte o valor x para o tipo T se a conversão for permitida. Muitas conversões de inteiro para inteiro não provocam nenhuma mudança no valor; elas simplesmente informam como o compilador deve interpretar um valor. Porém, uma conversão que restrinja um inteiro maior em um valor menor, ou uma conversão de inteiro para um número de ponto flutuante ou vice-versa podem alterar o valor ou provocar perda de precisão:

```
f := 3.141           // um float64
i := int(f)
fmt.Println(f, i)    // "3.141 3"
f = 1.99
fmt.Println(int(f))  // "1"
```

Conversões de ponto flutuante para inteiro descartam qualquer parte fracionária, truncando o resultado na direção de zero. Evite conversões em que o operando esteja fora do intervalo do tipo do alvo, pois o comportamento depende da implementação:

```
f := 1e100    // um float64
i := int(f)   // resultado depende da implementação
```

Inteiros literais de qualquer tamanho e tipo podem ser escritos como números decimais comuns, ou como números octais, se eles começarem com **0**, como em **0666**, ou como hexadecimais, se começarem com **0X** ou **0x**, como em **0xdeadbeef**. Dígitos hexa podem usar letra maiúscula ou minúscula. Atualmente, números octais parecem ser usados exclusivamente para um propósito – permissões de arquivo em sistemas POSIX – mas números hexadecimais são amplamente utilizados para enfatizar o padrão de bits de um número em vez de seu valor numérico.

Ao exibir números usando o pacote **fmt**, podemos controlar a base e o formato com os verbos **%d**, **%o** e **%x**, como mostra este exemplo:

```
o := 0666
fmt.Printf("%d %[1]o %#[1]o\n", o)    // "438 666 0666"
x := int64(0xdeadbeef)
fmt.Printf("%d %[1]x %#[1]x %#[1]X\n", x)
// Saída:
// 3735928559 deadbeef 0xdeadbeef 0XDEADBEEF
```

Observe o uso de dois truques de **fmt**. Normalmente, uma string de formatação de **Printf** contendo vários verbos **%** exigiria o mesmo número de operandos extras, mas os “advérbios” **[1]** após **%** dizem a **Printf** para usar o primeiro operando repetidamente. Em segundo lugar, o advérbio **#** para **%o** ou **%x** ou **%X** diz a **Printf** para gerar um prefixo **0** ou **0x** ou **0X**, respectivamente.

Runas literais são escritas como um caractere entre aspas simples. O exemplo mais simples é um caractere ASCII como **'a'**, mas é possível escrever qualquer ponto de código Unicode diretamente ou com escapes numéricos, como veremos em breve.

Runas são exibidas com **%c**, ou com **%q** caso se queira usar aspas:

```

ascii := 'a'
unicode := '国'
newline := '\n'
fmt.Printf("%d %[1]c %[1]q\n", ascii)    // "97 a 'a'"
fmt.Printf("%d %[1]c %[1]q\n", unicode) // "22269 国 '国'"
fmt.Printf("%d %[1]q\n", newline)       // "10 '\n'"

```

3.2 Números de ponto flutuante

Go oferece dois tamanhos de números de ponto flutuante: **float32** e **float64**. Suas propriedades aritméticas são governadas pelo padrão IEEE 754, implementado por todas as CPUs modernas.

Valores desses tipos numéricos variam de minúsculos para enormes. Os limites para valores de ponto flutuante podem ser encontrados no pacote **math**. A constante **math.MaxFloat32**, que é o maior valor de **float32**, é aproximadamente igual a **3.4e38**, e **math.MaxFloat64** é aproximadamente **1.8e308**. Os menores valores positivos são próximos de **1.4e-45** e **4.9e-324**, respectivamente.

Um **float32** oferece em torno de seis dígitos decimais de precisão, enquanto um **float64** oferece cerca de 15 dígitos; deve-se dar preferência a **float64** na maioria dos casos, pois cálculos com **float32** acumulam erros rapidamente, a menos que se tenha bastante cuidado, e o menor inteiro positivo que não pode ser representado de forma exata como um **float32** não é grande:

```

var f float32 = 16777216 // 1 << 24
fmt.Println(f == f+1)    // "true"!

```

Números de ponto flutuante podem ser escritos literalmente usando decimais, assim:

```

const e = 2.71828 // (aproximadamente)

```

Dígitos podem ser omitidos antes do ponto decimal (**.707**) ou depois dele (**1.**). Números muito pequenos ou muito grandes são escritos melhor em notação científica, com a letra **e** ou **E** antes do expoente decimal:

```

const Avogadro = 6.02214129e23
const Planck   = 6.62606957e-34

```

Valores de ponto flutuante são convenientemente exibidos com o verbo `%g` de `Printf`, que escolhe a representação mais compacta com a precisão adequada, mas para tabelas de dados, as formas `%e` (expoente) ou `%f` (sem expoente) podem ser mais apropriadas. Os três verbos permitem que a largura do campo e a precisão numérica sejam controladas.

```
for x := 0; x < 8; x++ {  
    fmt.Printf("x = %d eX = %8.3f\n", x, math.Exp(float64(x)))  
}
```

O código anterior exibe as potências de *e* com três dígitos decimais de precisão, alinhadas em um campo de oito caracteres:

```
x = 0   eX =    1.000  
x = 1   eX =    2.718  
x = 2   eX =    7.389  
x = 3   eX =   20.086  
x = 4   eX =   54.598  
x = 5   eX =  148.413  
x = 6   eX = 403.429  
x = 7   eX =1096.633
```

Além de uma coleção grande de funções matemáticas comuns, o pacote `math` tem funções para criar e identificar os valores especiais definidos pelo IEEE 754: os infinitos positivo e negativo, que representam números de magnitude excessiva e o resultado da divisão por zero, e NaN (“Not a Number”), que é o resultado de operações matematicamente duvidosas como `0/0` ou `Sqrt(-1)`.

```
var z float64  
fmt.Println(z, -z, 1/z, -1/z, z/z) // "0 -0 +Inf -Inf NaN"
```

A função `math.IsNaN` testa se seu argumento é um valor not-a-number e `math.NaN` devolve esse valor. É tentador usar NaN como um valor de sentinela em um processamento numérico, mas testar se o resultado de um cálculo específico é igual a NaN é muito perigoso, pois qualquer comparação com NaN *sempre* resulta em `false` (exceto `!=` que é sempre a negação de `==`):

```
nan := math.NaN()
fmt.Println(nan == nan, nan < nan, nan > nan) // "false false false"
```

Se uma função que devolva um número de ponto flutuante falhar, é melhor informar a falha separadamente, assim:

```
func compute() (value float64, ok bool) {
    // ...
    if failed {
        return 0, false
    }
    return result, true
}
```

O próximo programa mostra o processamento de gráficos de ponto flutuante. Ele desenha uma função de duas variáveis $z = f(x, y)$ como uma superfície de malha (wire mesh) 3D, usando SVG (Scalable Vector Graphics, ou Gráficos Vetoriais Escaláveis), que é uma notação XML padrão para desenho vetorial. A figura 3.1 mostra um exemplo de sua saída para a função $\sin(r)/r$, em que r é $\text{sqrt}(x*x+y*y)$.

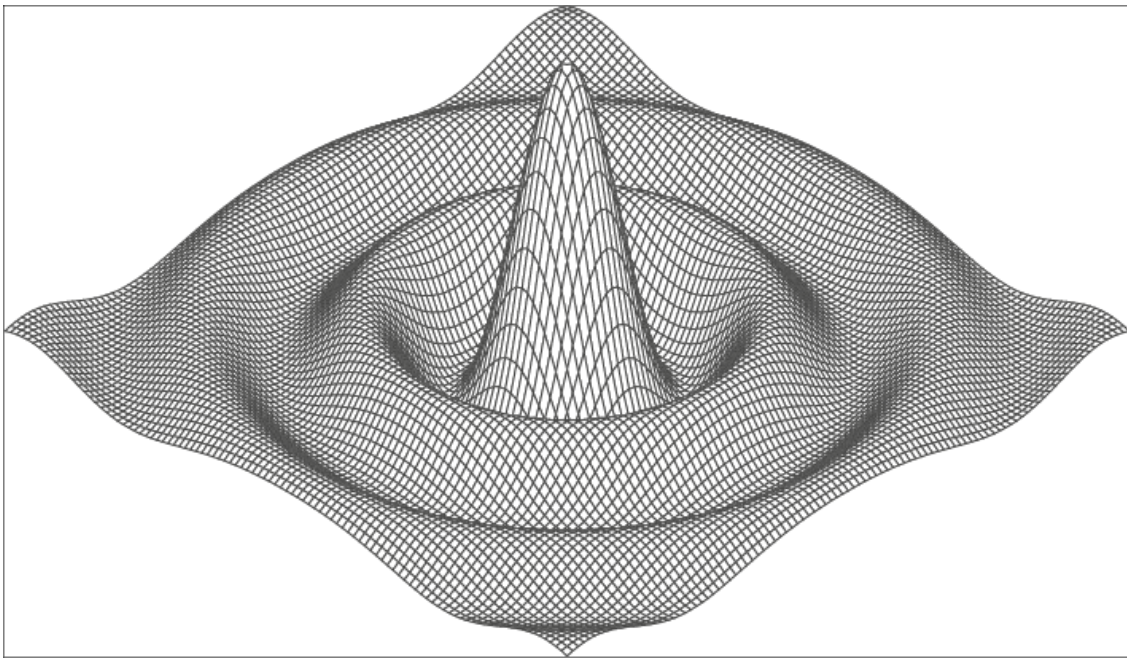


Figura 3.1 – Desenho da superfície da função $\sin(r)/r$.

gopl.io/ch3/surface

```
// Surface calcula uma renderização SVG de uma função de superfície 3D.
package main
```

```

import (
    "fmt"
    "math"
)

const (
    width, height = 600, 320    // tamanho do canvas em pixels
    cells         = 100        // número de células da grade
    xyrange       = 30.0       // intervalos dos eixos (-xyrange..+xyrange)
    xyscale       = width / 2 / xyrange // pixels por unidade x ou y
    zscale        = height * 0.4 // pixels por unidade z
    angle         = math.Pi / 6 // ângulo dos eixos x, y (=30°)
)

var sin30, cos30 = math.Sin(angle), math.Cos(angle) // seno(30°), cosseno(30°)

func main() {
    fmt.Printf("<svg xmlns='http://www.w3.org/2000/svg' "+
        "style='stroke: grey; fill: white; strokewidth: 0.7' "+
        "width='%d' height='%d'>", width, height)
    for i := 0; i < cells; i++ {
        for j := 0; j < cells; j++ {
            ax, ay := corner(i+1, j)
            bx, by := corner(i, j)
            cx, cy := corner(i, j+1)
            dx, dy := corner(i+1, j+1)
            fmt.Printf("<polygon points='%g,%g %g,%g %g,%g %g,%g' />\n",
                ax, ay, bx, by, cx, cy, dx, dy)
        }
    }
    fmt.Println("</svg>")
}

func corner(i, j int) (float64, float64) {
    // Encontra o ponto (x,y) no canto da célula (i,j)
    x := xyrange * (float64(i)/cells - 0.5)
    y := xyrange * (float64(j)/cells - 0.5)

    // Calcula a altura z da superfície
    z := f(x, y)

    // Faz uma projeção isométrica de (x,y,z) sobre (sx,sy) do canvas SVG 2D
    sx := width/2 + (x-y)*cos30*xyscale
    sy := height/2 + (x+y)*sin30*xyscale - z*zscale
    return sx, sy
}

func f(x, y float64) float64 {

```

```

    r := math.Hypot(x, y) // distância de (0,0)
    return math.Sin(r) / r
}

```

Observe que a função **corner** devolve dois valores, que são as coordenadas do canto da célula.

A explicação de como o programa funciona exige apenas geometria básica, mas não há problemas em ignorá-la, pois a questão principal é mostrar o processamento com números de ponto flutuante. A essência do programa está em fazer o mapeamento entre três sistemas de coordenadas diferentes, como mostra a figura 3.2. O primeiro sistema é uma grade 2D de 100 x100 células identificadas por coordenadas inteiras (i , j), começando em (0, 0) no canto mais distante ao fundo. Desenhamos de trás para a frente para que polígonos em segundo plano possam ser ocultos pelos que estiverem em primeiro plano.

O segundo sistema de coordenadas é uma malha de coordenadas 3D (x , y , z) de números de ponto flutuante, em que x e y são funções lineares de i e j , transladados de modo que a origem está no centro, e escalados pela constante **xyrange**. A altura z é o valor da função de superfície $f(x, y)$.

O terceiro sistema de coordenadas é o canvas da imagem 2D, com (0, 0) no canto superior esquerdo. Pontos nesse plano são representados como (sx , sy). Usamos uma projeção isométrica para mapear cada ponto 3D (x , y , z) sobre o canvas 2D.

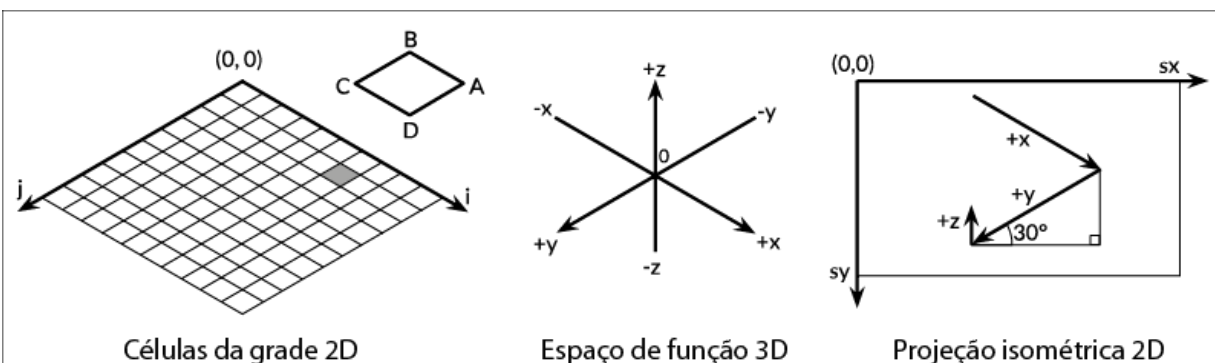


Figura 3.2 – Três sistemas de coordenadas diferentes.

Um ponto estará mais distante à direita no canvas quanto maior for seu valor x ou quanto *menor* for seu valor y . Um ponto aparecerá mais para

baixo no canvas quanto maior for seu valor x ou y e quanto menor for seu valor z . Os fatores de escala vertical e horizontal para x e y são derivados do seno e do cosseno de um ângulo de 30° . O fator de escala para z , igual a 0,4, é um parâmetro arbitrário.

Para cada célula na grade 2D, a função principal calcula as coordenadas no canvas da imagem dos quatro cantos do polígono $ABCD$, em que B corresponde a (i, j) e A , C e D são seus vizinhos e, então, gera uma instrução SVG para desenhá-lo.

Exercício 3.1: Se a função f devolver um valor `float64` não finito, o arquivo SVG conterá elementos `<polygon>` inválidos (embora muitos renderizadores SVG tratem essa situação com elegância). Modifique o programa para ignorar polígonos inválidos que forem gerados.

Exercício 3.2: Faça experimentos com visualizações de outras funções do pacote `math`. Você pode gerar padrões como caixa de ovo, morrinhos (moguls)¹ ou uma sela?

Exercício 3.3: Pinte cada polígono de acordo com sua altura, de modo que os picos tenham a cor vermelha (`#ff0000`) e os vales sejam azuis (`#0000ff`).

Exercício 3.4: Seguindo a abordagem do exemplo Lissajous na seção 1.7, crie um servidor web que calcule superfícies e escreva dados SVG ao cliente. O servidor deve definir o cabeçalho **Content-Type** assim:

```
w.Header().Set("Content-Type", "image/svg+xml")
```

(Esse passo não foi necessário no exemplo de Lissajous porque o servidor usa métodos heurísticos padrão para reconhecer formatos comuns como PNG a partir dos primeiros 512 bytes da resposta e gera o cabeçalho apropriado.) Permita que o cliente especifique valores como altura, largura e cor como parâmetros da requisição HTTP.

3.3 Números complexos

Go oferece dois tamanhos de números complexos, `complex64` e `complex128`, cujos componentes são `float32` e `float64`, respectivamente. A função embutida `complex` cria um número complexo a partir de seus

componentes real e imaginário, e as funções embutidas `real` e `imag` extraem esses componentes:

```
var x complex128 = complex(1, 2) // 1+2i
var y complex128 = complex(3, 4) // 3+4i
fmt.Println(x*y)                  // "(-5+10i)"
fmt.Println(real(x*y))            // "-5"
fmt.Println(imag(x*y))            // "10"
```

Se um número de ponto flutuante literal ou um inteiro decimal literal for seguido imediatamente de `i`, como em `3.141592i` ou `2i`, ele se torna um *imaginário literal*, representando um número complexo com um componente real igual a zero:

```
fmt.Println(1i * 1i) // "(-1+0i)",  $i^2 = -1$ 
```

De acordo com as regras da aritmética de constantes, as constantes complexas podem ser somadas a outras constantes numéricas (inteiro ou número de ponto flutuante, real ou imaginário), permitindo escrever números complexos naturalmente, como `1+2i` ou, de modo equivalente, `2i+1`. As declarações anteriores de `x` e `y` podem ser simplificadas:

```
x := 1 + 2i
y := 3 + 4i
```

Números complexos podem ser comparados para testar se são iguais com `==` e `!=`. Dois números complexos são iguais se suas partes reais forem iguais e suas partes imaginárias forem iguais.

O pacote `math/cmplx` oferece funções de biblioteca para trabalhar com números complexos, por exemplo, raiz quadrada de números complexos e funções de exponenciação.

```
fmt.Println(cmplx.Sqrt(-1)) // "(0+1i)"
```

O programa a seguir usa a aritmética com `complex128` para gerar um conjunto de Mandelbrot.

gopl.io/ch3/mandelbrot

```
// Mandelbrot gera uma imagem PNG do fractal de Mandelbrot.
package main

import (
    "image"
```

```

    "image/color"
    "image/png"
    "math/cmplx"
    "os"
)
func main() {
    const (
        xmin, ymin, xmax, ymax = -2, -2, +2, +2
        width, height           = 1024, 1024
    )

    img := image.NewRGBA(image.Rect(0, 0, width, height))
    for py := 0; py < height; py++ {
        y := float64(py)/height*(ymax-ymin) + ymin
        for px := 0; px < width; px++ {
            x := float64(px)/width*(xmax-xmin) + xmin
            z := complex(x, y)
            // Ponto (px, py) da imagem representa o valor complexo z
            img.Set(px, py, mandelbrot(z))
        }
    }
    png.Encode(os.Stdout, img) // NOTA: ignorando erros
}

func mandelbrot(z complex128) color.Color {
    const iterations = 200
    const contrast = 15

    var v complex128
    for n := uint8(0); n < iterations; n++ {
        v = v*v + z
        if cmplx.Abs(v) > 2 {
            return color.Gray{255 - contrast*n}
        }
    }
    return color.Black
}

```

Os dois loops aninhados iteram sobre cada ponto em uma imagem raster de tons de cinza de 1024x1024 que representa a porção de -2 a $+2$ do plano complexo. O programa testa se calcular repetidamente o quadrado e somar o número que o ponto representa, em algum momento, faz com que se “escape” do círculo de raio 2. Em caso afirmativo, o ponto é sombreado de acordo com o número de iterações necessário para escapar.

Caso contrário, o valor pertence ao conjunto de Mandelbrot, e o ponto permanece preto. Por fim, o programa desenha a imagem do fractal icônico codificada como PNG na saída-padrão, como mostra a figura 3.3.

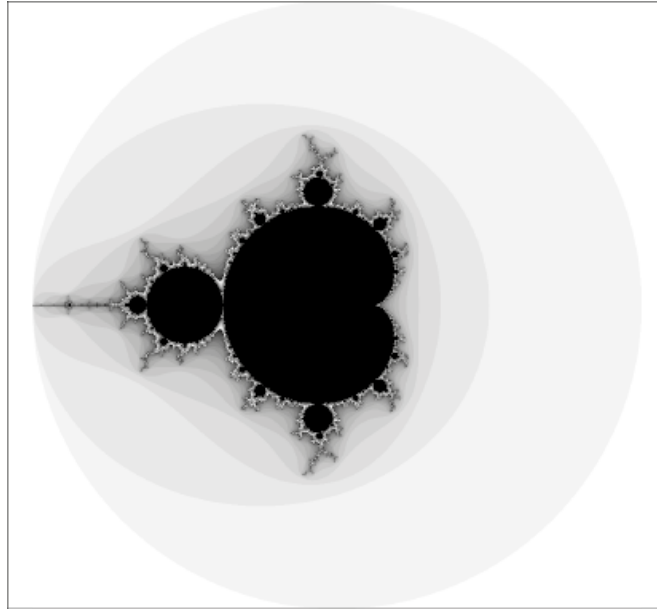


Figura 3.3 – O conjunto de Mandelbrot.

Exercício 3.5: Implemente o conjunto de Mandelbrot todo colorido usando a função `image.NewRGBA` e o tipo `color.RGBA` ou `color.YCbCr`.

Exercício 3.6: Superamostragem (supersampling) é uma técnica para reduzir o efeito de *pixelation*², calculando o valor da cor em vários pontos em cada pixel e tirando a média. O método mais simples é dividir cada pixel em quatro “subpixels”. Implemente isso.

Exercício 3.7: Outro fractal simples usa o método de Newton para encontrar soluções complexas a uma função como $z^4 - 1 = 0$. Sombreie cada ponto de partida de acordo com o número de iterações necessárias para se aproximar de uma das quatro raízes. Pinte cada ponto segundo a raiz da qual ele se aproxima.

Exercício 3.8: Renderizar fractais com níveis altos de zoom exige alta precisão aritmética. Implemente o mesmo fractal usando quatro representações numéricas diferentes: `complex64`, `complex128`, `big.Float` e `big.Rat`. (Os dois últimos tipos encontram-se no pacote `math/big`.)

Float usa números de ponto flutuante quaisquer, porém com precisão limitada; **Rat** usa números racionais com precisão ilimitada.) Como eles se comparam quanto ao desempenho e ao uso de memória? Em que níveis de zoom os artefatos de renderização tornam-se visíveis?

Exercício 3.9: Escreva um servidor web que renderize fractais e escreva os dados da imagem ao cliente. Permita que o cliente especifique os valores de *x*, *y* e de zoom como parâmetros da requisição HTTP.

3.4 Booleanos

Um valor do tipo **bool**, ou *booleano*, tem apenas dois valores possíveis: **true** e **false**. As condições em instruções **if** e **for** são booleanas, e os operadores de comparação como **==** e **<** geram um resultado booleano. O operador unário **!** é a negação lógica, portanto **!true** é **false**, ou poderíamos dizer que **(!true==false)==true**, embora, por questões de estilo, sempre simplificamos expressões booleanas redundantes como **x==true** para **x**.

Valores booleanos podem ser combinados com os operadores **&&** (E) e **||** (OU), que têm um comportamento de *curto-circuito*: se a resposta já estiver determinada pelo valor do operando esquerdo, o operando direito não é avaliado, tornando seguro escrever expressões como:

```
s != "" && s[0] == 'x'
```

em que **s[0]** causaria pânico se fosse aplicado a uma string vazia.

Como **&&** tem mais precedência que **||** (mnemônico: **&&** é a multiplicação booleana, **||** é a adição booleana), não são necessários parênteses para condições neste formato:

```
if 'a' <= c && c <= 'z' ||  
    'A' <= c && c <= 'Z' ||  
    '0' <= c && c <= '9' {  
    // ...letra ou dígito ASCII...  
}
```

Não há conversão implícita de um valor booleano para um valor numérico como 0 ou 1, ou vice-versa. É preciso usar um **if** explícito,

como em:

```
i := 0
if b {
    i = 1
}
```

Talvez valha a pena escrever uma função de conversão se essa operação for necessária com frequência:

```
// btoi devolve 1 se b é true e 0 se false
func btoi(b bool) int {
    if b {
        return 1
    }
    return 0
}
```

A operação inversa é tão simples que não justifica ter uma função, mas, por questões de simetria, temos:

```
// itob informa se i é diferente de zero
func itob(i int) bool { return i != 0 }
```

3.5 Strings

Uma string é uma sequência imutável de bytes. Strings podem conter qualquer dado, incluindo bytes com valor 0, mas normalmente elas contêm texto legível aos seres humanos. As strings de texto são convencionalmente interpretadas como sequências de pontos de código Unicode (runas) codificadas em UTF-8, que exploraremos em detalhes muito em breve.

A função embutida `len` devolve o número de bytes (não de runas) em uma string, e a operação de *índice* `s[i]` recupera o *i*-ésimo byte da string `s`, em que $0 \leq i < \text{len}(s)$.

```
s := "hello, world"
fmt.Println(len(s))      // "12"
fmt.Println(s[0], s[7])  // "104 119" ('h' e 'w')
```

Tentar acessar um byte fora desse intervalo resulta em pânico:

```
c := s[len(s)] // pânico: índice fora do intervalo
```

O *i*-ésimo byte de uma string não é necessariamente o *i*-ésimo *caractere* de uma string, pois a codificação UTF-8 de um ponto de código que não seja ASCII exige dois ou mais bytes. A forma de trabalhar com caracteres será discutida em breve.

A operação de *substring* `s[i:j]` produz uma nova string constituída dos bytes da string original, começando no índice *i* e continuando até o byte no índice *j*, mas sem incluí-lo. O resultado contém *j-i* bytes.

```
fmt.Println(s[0:5]) // "hello"
```

Novamente, ocorrerá um pânico se um dos índices estiver fora dos limites ou se *j* for menor que *i*.

Tanto o operando *i* pode ser omitido quanto o operando *j*, ou ambos, caso em que os valores default 0 (o início da string) e `len(s)` (seu fim) serão usados, respectivamente.

```
fmt.Println(s[:5]) // "hello"
fmt.Println(s[7:]) // "world"
fmt.Println(s[:])  // "hello, world"
```

O operador `+` cria uma nova string concatenando duas strings:

```
fmt.Println("goodbye" + s[5:]) // "goodbye, world"
```

Strings podem ser comparadas com os operadores de comparação como `==` e `<`; a comparação é feita byte a byte, portanto o resultado é a ordem lexicográfica natural³.

Valores de string são imutáveis: a sequência de bytes contida em um valor de string jamais pode ser alterada, embora, é claro, possamos atribuir um novo valor a uma *variável* do tipo string. Para concatenar uma string em outra, por exemplo, podemos escrever:

```
s := "left foot"
t := s
s += ", right foot"
```

Isso não modifica a string armazenada originalmente em `s`, mas faz `s` armazenar a nova string composta pela instrução `+=`; enquanto isso, `t` continua armazenando a string antiga.

```
fmt.Println(s) // "left foot, right foot"
```

```
fmt.Println(t) // "left foot"
```

Como as strings são imutáveis, construções que tentam modificar os dados de uma string in-place (no próprio lugar) não são permitidas:

```
s[0] = 'L' // erro de compilação: não é possível fazer uma atribuição a s[0]
```

Imutabilidade implica que é seguro que duas cópias de uma string compartilhem a mesma memória subjacente, fazendo com que não seja custoso copiar strings de qualquer tamanho. De modo semelhante, uma string `s` e uma substring como `s[7:]` podem compartilhar os mesmos dados de forma segura, portanto a operação de substring também não é custosa. Nenhuma área nova de memória é alocada nesses casos. A figura 3.4 mostra a organização de uma string e de duas de suas substrings compartilhando o mesmo array de bytes subjacente.

3.5.1 Strings literais

Um valor de string pode ser escrito como uma *string literal*, isto é, uma sequência de bytes entre aspas duplas:

```
"Hello, 世界"
```

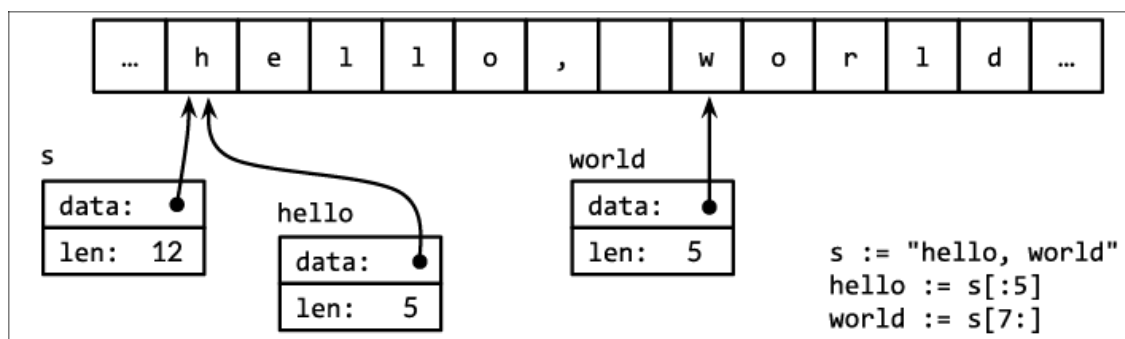


Figura 3.4 – A string “hello, world” e duas substrings.

Como os arquivos-fontes de Go são sempre codificados em UTF-8 e as strings de texto de Go são convencionalmente interpretadas como UTF-8, podemos incluir pontos de código Unicode em strings literais.

Em uma string literal entre aspas duplas, *sequências de escape* que começam com uma barra invertida `\` podem ser usadas para inserir valores arbitrários de bytes na string. Um conjunto de escapes representa códigos de controle ASCII como quebra de linha, carriage return e

tabulação:

\a “alerta” ou sino

\b backspace

\f form feed

\n quebra de linha

\r carriage return

\t tabulação

\v tabulação vertical

\' aspas simples (somente na runa literal '\')

\" aspas duplas (somente em literais "...")

\\ barra invertida

Bytes arbitrários também podem ser incluídos em strings literais usando escapes hexadecimais ou octais. Um *escape hexadecimal* é escrito como \xhh, com exatamente dois dígitos hexadecimais *h* (em letra maiúscula ou minúscula). Um *escape octal* é escrito como \ooo, com exatamente três dígitos octais *o* (de 0 a 7), sem exceder \377. Ambos representam um único byte com o valor especificado. Mais adiante veremos como codificar pontos de código Unicode numericamente em strings literais.

Uma *string literal bruta* (*raw literal string*) é escrita como ``...``, usando crases no lugar de aspas duplas. Em uma string literal bruta, nenhuma sequência de escape é processada; o conteúdo é interpretado literalmente, incluindo barras invertidas e quebras de linha, portanto uma string literal bruta pode ocupar diversas linhas no código-fonte do programa. O único processamento é a remoção dos carriage returns para que o valor da string seja igual em todas as plataformas, incluindo naquelas que, convencionalmente, colocam carriage returns em arquivos-texto.

Strings literais brutas são uma maneira conveniente de escrever expressões regulares, que tendem a ter muitas barras invertidas. Elas também são úteis para templates HTML, literais JSON, mensagens de uso de comandos e informações desse tipo que, com frequência, ocupam várias

linhas.

```
const GoUsage = `Go is a tool for managing Go source code.  
Usage:  
    go command [arguments]  
...`
```

3.5.2 Unicode

Antigamente a vida era simples e havia, pelo menos de um ponto de vista estreito, apenas um conjunto de caracteres para se lidar: ASCII (American Standard Code for Information Interchange, ou Código americano padrão para intercâmbio de informações). O ASCII, ou mais precisamente, o US-ASCII, usa 7 bits para representar 128 “caracteres”: as letras maiúsculas e minúsculas usadas em inglês, os dígitos e vários caracteres de pontuação e de controle de dispositivos. Durante um bom período do início da computação isso era adequado, mas deixou uma fração bem grande da população mundial incapaz de usar seus próprios sistemas de escrita nos computadores. Com o crescimento da internet, dados em uma variedade de línguas tornaram-se muito mais comuns. Como podemos lidar com essa grande variedade e, se possível, com eficiência?

A resposta está no Unicode (unicode.org), que reúne todos os caracteres de todos os sistemas de escrita do mundo, além de acentos e outras marcas diacríticas, códigos de controle como tabulação e carriage return e muitos caracteres esotéricos, e atribui a cada item um número padrão chamado ponto de código Unicode (Unicode code point), ou na terminologia de Go, uma *runa* (rune).

A versão 8 do Unicode define pontos de código para mais de 120 mil caracteres em mais de cem línguas e formas de escrita. Como eles são representados em programas e dados de computador? O tipo de dado natural para armazenar uma única runa é `int32`, e é isso que Go usa; o sinônimo `rune` é usado exatamente para esse propósito.

Poderíamos representar uma sequência de runas como uma sequência de valores `int32`. Nessa representação, que se chama UTF-32 ou UCS-4, a

codificação de cada ponto de código Unicode tem o mesmo tamanho, isto é, 32 bits. É simples e uniforme, mas usa muito mais espaço que o necessário, pois a maior parte dos textos legíveis no computador está em ASCII, que exige apenas 8 bits ou um byte por caractere. Todos os caracteres amplamente usados totalizam menos de 65.536, portanto caberiam em 16 bits. Podemos fazer algo melhor?

3.5.3 UTF-8

O UTF-8 é uma codificação de tamanho variável de pontos de código Unicode na forma de bytes. O UTF-8 foi inventado por Ken Thompson e Rob Pike, dois dos criadores de Go, e atualmente é um padrão do Unicode. Esse padrão usa entre 1 e 4 bytes para representar cada runa, mas somente um byte para caracteres ASCII e apenas 2 ou 3 bytes para a maioria das runas de uso comum. Os bits de alta ordem do primeiro byte da codificação de uma runa informam quantos bytes existem na sequência. Um 0 no bit de mais alta ordem indica ASCII de 7 bits, em que cada runa ocupa apenas um byte, portanto é idêntico ao ASCII convencional. Bits de alta ordem iguais a 110 indicam que a runa ocupa 2 bytes; o segundo byte começa com 10. Runas maiores têm codificações análogas.

0xxxxxxx	runas 0-127	(ASCII)
110xxxxx 10xxxxxx	128-2047	(valores <128 não usados)
1110xxxx 10xxxxxx 10xxxxxx	2048-65535	(valores <2048 não usados)
11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	65536-0x10ffff	(outros valores não usados)

Uma codificação de tamanho variável impossibilita a indexação direta para acessar o n -ésimo caractere de uma string, mas o UTF-8 tem muitas propriedades desejáveis para compensar. A codificação é compacta, compatível com ASCII e sincroniza-se por si só: é possível encontrar o início de um caractere retrocedendo não mais do que três bytes. Também é um código de prefixo, portanto pode ser decodificado da esquerda para a direita, sem ambiguidade ou a necessidade de olhar adiante. Nenhuma codificação de runa é uma substring de outra, nem mesmo de uma sequência de outras, assim você pode procurar uma runa apenas

pesquisando seus bytes, sem se preocupar com o contexto anterior. A ordem lexicográfica dos bytes é igual à ordem dos pontos de código Unicode, desta forma a ordenação de UTF-8 funciona naturalmente. Não há bytes NUL (zero) embutidos, o que é conveniente para linguagens de programação que usam NUL para terminar strings.

Os arquivos-fonte de Go são sempre codificados em UTF-8, e o UTF-8 é a codificação preferida para strings de texto manipuladas por programas Go. O pacote **unicode** oferece funções para trabalhar com runas individuais (por exemplo, distinguir letras de números ou converter uma letra maiúscula para uma letra minúscula) e, o pacote **unicode/utf8** oferece funções para codificar e decodificar runas como bytes usando UTF-8.

Muitos caracteres Unicode são difíceis de digitar em um teclado ou distinguir visualmente de outros com aparência semelhante; alguns são até mesmo invisíveis. Escapes Unicode em strings literais de Go permitem especificar caracteres pelo valor numérico de seu ponto de código. Existem duas formas, `\uhhhh` para um valor de 16 bits e `\Uhhhhhhhh` para um valor de 32 bits, em que cada *h* é um dígito hexadecimal; a necessidade da forma de 32 bits não é muito frequente. Cada uma representa a codificação UTF-8 do ponto de código especificado. Por exemplo, as strings literais a seguir representam a mesma string de seis bytes:

```
"世界"  
"\xe4\xb8\x96\xe7\x95\x8c"  
"\u4e16\u754c"  
"\U00004e16\U0000754c"
```

As três sequências de escape anteriores oferecem notações alternativas para a primeira string, mas os valores que elas representam são idênticos.

Escapes Unicode também podem ser usados em runas literais. Estes três literais são equivalentes:

```
'世' '\u4e16' '\U00004e16'
```

Uma runa cujo valor seja menor que 256, pode ser escrita com um único escape hexadecimal, como `'\x41'` para `'A'`, mas para valores maiores um

escape `\u` ou `\U` deve ser usado. Consequentemente, `'\xe4\xb8\x96'` não é uma runa literal permitida, apesar de esses três bytes serem uma codificação UTF-8 válida de um único ponto de código.

Graças às boas propriedades do UTF-8, muitas operações com strings não exigem decodificação. Podemos testar se uma string contém outra como prefixo:

```
func HasPrefix(s, prefix string) bool {
    return len(s) >= len(prefix) && s[:len(prefix)] == prefix
}
```

ou como sufixo:

```
func HasSuffix(s, suffix string) bool {
    return len(s) >= len(suffix) && s[len(s)-len(suffix):] == suffix
}
```

ou como uma substring:

```
func Contains(s, substr string) bool {
    for i := 0; i < len(s); i++ {
        if HasPrefix(s[i:], substr) {
            return true
        }
    }
    return false
}
```

usando a mesma lógica utilizada com bytes puros em texto codificado em UTF-8. Isso não vale para outras codificações. (As funções anteriores foram extraídas do pacote **strings**, embora sua implementação de **Contains** utilize uma técnica de hashing para fazer pesquisas com mais eficiência.)

Por outro lado, se realmente estivermos interessados nos caracteres Unicode individuais, devemos usar outros mecanismos. Considere a string de nosso primeiro exemplo, que inclui dois caracteres do leste asiático. A figura 3.5 mostra sua representação na memória. A string contém 13 bytes, mas ao ser interpretada como UTF-8, ela codifica somente nove pontos de código ou runas:

```
import "unicode/utf8"
```

```
s := "Hello, 世界"
fmt.Println(len(s))           // "13"
fmt.Println(utf8.RuneCountInString(s)) // "9"
```

Para processar esses caracteres, precisamos de um decodificador de UTF-8. O pacote **unicode/utf8** oferece um que podemos usar assim:

```
for i := 0; i < len(s); {
    r, size := utf8.DecodeRuneInString(s[i:])
    fmt.Printf("%d\t%c\n", i, r)
    i += size
}
```

Cada chamada a **DecodeRuneInString** devolve **r**, que é a runa propriamente dita, e **size**, que é o número de bytes usados pela codificação UTF-8 de **r**. O tamanho é usado para atualizar o índice de bytes **i** da próxima runa da string. Porém, isso não é nada prático, e precisamos de loops desse tipo o tempo todo. Felizmente, o loop **range** de Go, quando é aplicado a uma string, executa a decodificação de UTF-8 implicitamente. A saída do loop a seguir também está mostrada na figura 3.5; observe como o índice salta mais de uma unidade para cada runa diferente de ASCII.

```
for i, r := range "Hello, 世界" {
    fmt.Printf("%d\t%q\t%d\n", i, r, r)
}
```

Poderíamos usar um loop **range** simples para contar o número de runas em uma string, desta maneira:

```
n := 0
for _, _ = range s {
    n++
}
```

Como ocorre com as outras formas do loop **range**, podemos omitir as variáveis de que não precisamos:

```
n := 0
for range s {
    n++
}
```

Ou podemos simplesmente chamar **utf8.RuneCountInString(s)**.

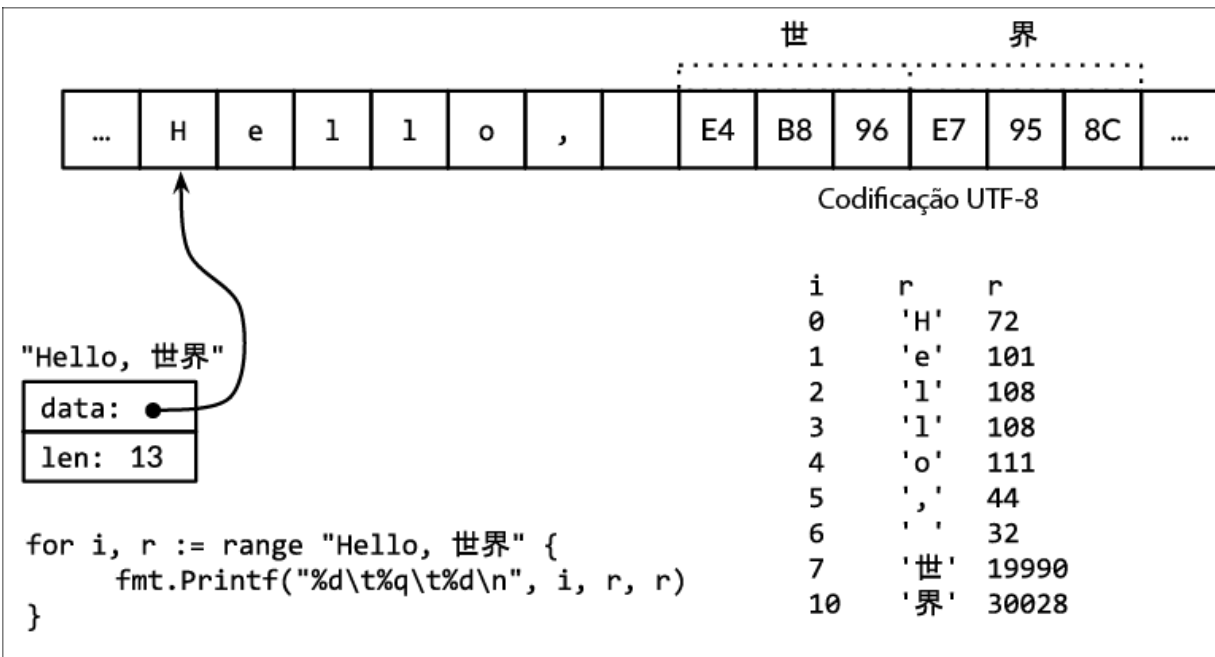


Figura 3.5 – Um loop **range** decodifica uma string codificada em UTF-8.

Mencionamos antes que strings de texto serem interpretadas como sequências de pontos de código Unicode codificadas como UTF-8 é mais uma questão de convenção em Go, mas para o uso correto de loops **range** em strings, é mais que uma convenção – é uma necessidade. O que acontece se percorrermos uma string contendo dados binários quaisquer ou, se for o caso, dados UTF-8 contendo erros?

Sempre que um decodificador de UTF-8, seja explícito em uma chamada a **utf8.DecodeRuneInString** ou implícito em um loop **range**, consome um byte de entrada inesperado, ele gera um caractere Unicode especial chamado *caractere de substituição*, `'\uFFFD'`, que normalmente é exibido como um ponto de interrogação branco dentro de uma caixa hexagonal ou em forma de losango . Quando um programa se depara com esse valor de runa, geralmente é sinal de que alguma parte do sistema que gerou os dados da string foi descuidada em seu tratamento de codificação de texto.

O UTF-8 é excepcionalmente conveniente como formato de intercâmbio de dados, mas em um programa, as runas podem ser mais convenientes, pois elas têm tamanho uniforme e, desse modo, são facilmente indexadas

em arrays e em fatias.

Uma conversão `[]rune` aplicada a uma string codificada em UTF-8 devolve a sequência de pontos de código Unicode codificada pela string:

```
// a palavra "program" em japonês, em katakana
s := "プログラム"
fmt.Printf("% x\n", s) // "e3 83 97 e3 83 ad e3 82 b0 e3 83 a9 e3 83 a0"
r := []rune(s)
fmt.Printf("%x\n", r) // "[30d7 30ed 30b0 30e9 30e0]"
```

(O verbo `% x` no primeiro `Printf` insere um espaço entre cada par de dígitos hexa.)

Se uma fatia de runas for convertida em uma string, uma concatenação das codificações UTF-8 de cada runa será gerada:

```
fmt.Println(string(r)) // "プログラム"
```

Converter um valor inteiro em uma string interpreta o inteiro como um valor de runa e produz a representação UTF-8 dessa runa:

```
fmt.Println(string(65)) // "A", e não "65"
fmt.Println(string(0x4eac)) // "京"
```

Se a runa for inválida, o caractere de substituição é usado em seu lugar:

```
fmt.Println(string(1234567)) // "�"
```

3.5.4 Strings e fatias de bytes

Quatro pacotes padrões são particularmente importantes para manipulação de strings: **bytes**, **strings**, **strconv** e **unicode**. O pacote **strings** oferece muitas funções para procurar, substituir, comparar, cortar, separar e juntar strings.

O pacote **bytes** tem funções similares para manipular fatias de bytes, de tipo `[]byte`, que compartilham algumas propriedades com as strings. Pelo fato de as strings serem imutáveis, criar strings de forma incremental pode envolver muita alocação e cópia. Nesses casos, é mais eficiente usar o tipo **bytes.Buffer**, que mostraremos em breve.

O pacote **strconv** oferece funções para converter valores booleanos, inteiros e de ponto flutuante de e para suas representações em strings,

além de funções para inserir e remover aspas de strings.

O pacote **unicode** oferece funções como **IsDigit**, **IsLetter**, **IsUpper** e **IsLower** para classificar runas. Cada função aceita uma única runa como argumento e devolve um booleano. Funções de conversão como **ToUpper** e **ToLower** convertem uma runa no tipo de letra especificado – maiúsculo ou minúsculo – se ela for uma letra. Todas essas funções usam as categorias-padrão do Unicode para letras, dígitos e assim por diante. O pacote **strings** tem funções semelhantes, também chamadas **ToUpper** e **ToLower**, que devolvem uma nova string com a transformação especificada aplicada a cada caractere da string original.

A função **basename** a seguir foi inspirada no utilitário de shell do Unix de mesmo nome. Em nossa versão, **basename(s)** remove qualquer prefixo de **s** que pareça ser um path do sistema de arquivos, com componentes separados por barras, e remove qualquer sufixo que pareça ser um tipo de arquivo:

```
fmt.Println(basename("a/b/c.go")) // "c"
fmt.Println(basename("c.d.go"))   // "c.d"
fmt.Println(basename("abc"))      // "abc"
```

A primeira versão de **basename** faz todo o trabalho sem a ajuda de bibliotecas:

gopl.io/ch3/basename1

```
// basename remove componentes de diretório e um .sufixo.
// exemplos: a => a, a.go => a, a/b/c.go => c, a/b.c.go => b.c
func basename(s string) string {
    // Descarta a última '/' e tudo que estiver antes
    for i := len(s) - 1; i >= 0; i-- {
        if s[i] == '/' {
            s = s[i+1:]
            break
        }
    }
    // Preserva tudo que estiver antes do último '.'
    for i := len(s) - 1; i >= 0; i-- {
        if s[i] == '.' {
            s = s[:i]
            break
        }
    }
}
```

```

    }
}
return s
}

```

Uma versão mais simples usa a função de biblioteca `strings.LastIndex`:

gopl.io/ch3/basename2

```

func basename(s string) string {
    slash := strings.LastIndex(s, "/") // -1 se "/" não for encontrada
    s = s[slash+1:]
    if dot := strings.LastIndex(s, "."); dot >= 0 {
        s = s[:dot]
    }
    return s
}

```

Os pacotes `path` e `path/filepath` oferecem um conjunto mais genérico de funções para manipular nomes hierárquicos. O pacote `path` trabalha com paths delimitados por barras em qualquer plataforma. Não deve ser usado para nomes de arquivo, mas é apropriado para outros domínios, como o componente de path de um URL. Em comparação, `path/filepath` manipula nomes de arquivo usando as regras da plataforma host, por exemplo, `/foo/bar` para POSIX ou `c:\foo\bar` no Microsoft Windows.

Vamos continuar com outro exemplo de substring. A tarefa é tomar uma representação em string de um inteiro, por exemplo `"12345"`, e inserir vírgulas a cada três posições, como em `"12,345"`. Essa versão só funciona para inteiros; tratar números de ponto flutuante será deixado como exercício ao leitor.

gopl.io/ch3/comma

```

// comma insere vírgulas em uma string que representa um inteiro decimal
// não negativo
func comma(s string) string {
    n := len(s)
    if n <= 3 {
        return s
    }
    return comma(s[:n-3]) + "," + s[n-3:]
}

```

```
}
```

O argumento de **comma** é uma string. Se seu tamanho for menor ou igual a 3, nenhuma vírgula será necessária. Caso contrário, **comma** chama a si mesma recursivamente, com uma substring constituída de todos os caracteres exceto os três últimos, e concatena uma vírgula e os três últimos caracteres ao resultado da chamada recursiva.

Uma string contém um array de bytes que, depois de criada, é imutável. Por outro lado, os elementos de uma fatia de bytes podem ser livremente modificados.

Strings podem ser convertidas em fatias de bytes e vice-versa:

```
s := "abc"
b := []byte(s)
s2 := string(b)
```

Conceitualmente, a conversão **[]byte(s)** aloca um novo array de bytes que armazena uma cópia dos bytes de **s**, e produz uma fatia que referencia esse array como um todo. Um compilador otimizado pode ser capaz de evitar a alocação e a cópia em alguns casos, mas, em geral, a cópia é necessária para garantir que os bytes de **s** permaneçam inalterados, mesmo que os bytes de **b** sejam subsequentemente modificados. A conversão de fatia de bytes de volta para string com **string(b)** também cria uma cópia para garantir a imutabilidade da string **s2** resultante.

Para evitar conversões e alocação de memória desnecessária, muitas das funções utilitárias do pacote **bytes** têm versões paralelas de suas contrapartidas do pacote **strings**. Por exemplo, a seguir estão meia dúzia de funções de **strings**:

```
func Contains(s, substr string) bool
func Count(s, sep string) int
func Fields(s string) []string
func HasPrefix(s, prefix string) bool
func Index(s, sep string) int
func Join(a []string, sep string) string
```

e as funções correspondentes de **bytes**:

```

func Contains(b, subslice []byte) bool
func Count(s, sep []byte) int
func Fields(s []byte) [][]byte
func HasPrefix(s, prefix []byte) bool
func Index(s, sep []byte) int
func Join(s [][]byte, sep []byte) []byte

```

A única diferença é que as strings foram substituídas por fatias de bytes.

O pacote **bytes** oferece o tipo **Buffer** para manipulação eficiente das fatias de bytes. Um **Buffer** começa vazio, mas cresce à medida que dados de tipos como **string**, **byte** e **[]byte** são escritos nele. Como mostra o exemplo a seguir, uma variável **bytes.Buffer** não exige inicialização porque seu valor zero é utilizável:

gopl.io/ch3/printints

```

// intsToString é como fmt.Sprint(values), mas acrescenta vírgulas.
func intsToString(values []int) string {
    var buf bytes.Buffer
    buf.WriteByte '['
    for i, v := range values {
        if i > 0 {
            buf.WriteString(", ")
        }
        fmt.Fprintf(&buf, "%d", v)
    }
    buf.WriteByte ']'
    return buf.String()
}
func main() {
    fmt.Println(intsToString([]int{1, 2, 3})) // "[1, 2, 3]"
}

```

Quando concatenar a codificação UTF-8 de uma runa qualquer em um **bytes.Buffer**, é melhor usar o método **WriteRune** de **bytes.Buffer**, mas não há problemas em usar **WriteByte** para caracteres ASCII como '[' e ']'.

O tipo **bytes.Buffer** é extremamente versátil; quando discutirmos interfaces no capítulo 7, veremos como ele pode ser usado como substituto para um arquivo sempre que uma função de E/S exigir um destino para os bytes (**io.Writer**), como o **Fprintf** anterior, ou uma

fonte de bytes (`io.Reader`).

Exercício 3.10: Escreva uma versão não recursiva de `comma` usando `bytes.Buffer` no lugar de concatenação de strings.

Exercício 3.11: Melhore `comma` de modo que ela trate corretamente números de ponto flutuante e um sinal opcional.

Exercício 3.12: Escreva uma função que informe se duas strings são anagramas uma da outra, isto é, se elas contêm as mesmas letras em ordem diferente.

3.5.5 Conversões entre strings e números

Além das conversões entre strings, runas e bytes, com frequência é necessário converter entre valores numéricos e suas representações em string. Isso é feito com funções do pacote `strconv`.

Para converter um inteiro em uma string, uma opção é usar `fmt.Sprintf`; outra é usar a função `strconv.Itoa` (“inteiro para ASCII”):

```
x := 123
y := fmt.Sprintf("%d", x)
fmt.Println(y, strconv.Itoa(x)) // "123 123"
```

`FormatInt` e `FormatUint` podem ser usadas para formatar números em uma base diferente:

```
fmt.Println(strconv.FormatInt(int64(x), 2)) // "1111011"
```

Os verbos `%b`, `%d`, `%o` e `%x` de `fmt.Printf` em geral são mais convenientes que as funções `Format`, especialmente se quisermos incluir informações adicionais além do número:

```
s := fmt.Sprintf("x=%b", x) // "x=1111011"
```

Para interpretar uma representação em string como um inteiro, use as funções de `strconv`: `Atoi`, `ParseInt` ou `ParseUint` para inteiros sem sinal:

```
x, err := strconv.Atoi("123")           // x é um int
y, err := strconv.ParseInt("123", 10, 64) // base 10, até 64 bits
```

O terceiro argumento de `ParseInt` determina o tamanho do tipo inteiro

em que o resultado deve caber; por exemplo, 16 quer dizer `int16`, e o valor especial 0 implica `int`. Qualquer que seja o caso, o tipo do resultado `y` é sempre `int64`, que você pode então converter para um tipo menor.

Às vezes, `fmt.Scanf` é útil para parse de entradas constituídas de combinações organizadas de strings e números, todos em uma só linha, mas ela pode ser inflexível, especialmente quando tratar entradas incompletas ou irregulares.

3.6 Constantes

Constantes são expressões cujos valores são conhecidos do compilador; é garantido que sua avaliação ocorre em tempo de compilação, e não em tempo de execução. O tipo subjacente de toda constante é um tipo básico: booleano, string ou número.

Uma declaração `const` define valores nomeados que se parecem sintaticamente com variáveis, mas cujos valores são constantes, o que evita alterações acidentais (ou maliciosas) durante a execução do programa. Por exemplo, uma constante é mais apropriada que uma variável para uma constante matemática como `pi`, pois seu valor não mudará:

```
const pi = 3.14159    // aproximadamente; math.Pi é uma aproximação melhor
```

Como ocorre com as variáveis, uma sequência de constantes pode estar em uma só declaração; isso será apropriado para um grupo de valores relacionados:

```
const (  
    e = 2.71828182845904523536028747135266249775724709369995957496696763  
    pi = 3.14159265358979323846264338327950288419716939937510582097494459  
)
```

Muitos cálculos com constantes podem ser totalmente avaliados em tempo de compilação, reduzindo o trabalho necessário em tempo de execução e possibilitando outras otimizações pelo compilador. Erros normalmente detectados em tempo de execução podem ser informados em tempo de compilação quando seus operandos são constantes, por

exemplo, divisão de inteiros por zero, indexação de strings fora do intervalo e qualquer operação com números de ponto flutuante que resulte em um valor infinito.

Os resultados de todas as operações aritméticas, lógicas e de comparação aplicadas a operandos que sejam constantes são, eles mesmos, constantes, assim como os resultados de conversões e de chamadas a determinadas funções embutidas como `len`, `cap`, `real`, `imag`, `complex` e `unsafe.Sizeof` (seção 13.1).

Como seus valores são conhecidos pelo compilador, expressões constantes podem aparecer em tipos, especificamente como o tamanho de um tipo array:

```
const IPv4Len = 4

// parseIPv4 faz parse de um endereço IPv4 (d.d.d.d)
func parseIPv4(s string) IP {
    var p [IPv4Len]byte
    // ...
}
```

A declaração de uma constante pode especificar um tipo, assim como um valor, mas, na ausência de um tipo explícito, ele é inferido a partir da expressão do lado direito. No código a seguir, `time.Duration` é um tipo nomeado cujo tipo subjacente é `int64`, e `time.Minute` é uma constante desse tipo. Ambas as constantes declaradas a seguir, portanto, têm o tipo `time.Duration` também, como mostrado por `%T`:

```
const noDelay time.Duration = 0
const timeout = 5 * time.Minute
fmt.Printf("%T %[1]v\n", noDelay)    // "time.Duration 0"
fmt.Printf("%T %[1]v\n", timeout)    // "time.Duration 5m0s"
fmt.Printf("%T %[1]v\n", time.Minute) // "time.Duration 1m0s"
```

Quando uma sequência de constantes é declarada como um grupo, a expressão do lado direito pode ser omitida para todos, exceto para o primeiro do grupo, o que quer dizer que a expressão anterior e seu tipo devem ser usados novamente. Por exemplo:

```
const (
    a = 1
```

```

    b
    c = 2
    d
)
fmt.Println(a, b, c, d)    // "1 1 2 2"

```

Isso não é muito útil se a expressão do lado direito copiada implicitamente sempre for avaliada da mesma maneira. Mas e se ela puder variar? Isso nos leva ao **iota**.

3.6.1 Gerador de constantes **iota**

Uma declaração **const** pode usar o *gerador de constantes* **iota** a fim de criar uma sequência de valores relacionados sem escrever todos eles explicitamente. Em uma declaração **const**, o valor de **iota** começa em zero e é incrementado de um a cada item da sequência.

A seguir, apresentamos um exemplo do pacote **time**, que define constantes nomeadas do tipo **Weekday** para os dias da semana, começando com zero para **Sunday**. Esses tipos muitas vezes são chamados de *enumerações*, ou *enums*, abreviadamente.

```

type Weekday int
const (
    Sunday Weekday = iota
    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
    Saturday
)

```

Esse código declara **Sunday** como 0, **Monday** como 1, e assim sucessivamente.

Também podemos usar **iota** em expressões mais complexas, como no próximo exemplo do pacote **net**, em que cada um dos 5 bits menos significativos de um inteiro sem sinal recebe um nome e uma interpretação booleana distintos:


```

type Flags uint
const (
    FlagUp Flags = 1 << iota    // está ativo
    FlagBroadcast          // suporta recurso de acesso broadcast
    FlagLoopback           // é uma interface loopback
    FlagPointToPoint       // pertence a um link ponto a ponto
    FlagMulticast           // suporta recurso de acesso multicast
)

```

Conforme **iota** incrementa, cada constante recebe o valor **1 << iota**, que é avaliado como potências sucessivas de dois, cada uma correspondendo a um único bit. Podemos usar essas constantes em funções que testam, definem ou limpam um ou mais desses bits:

gopl.io/ch3/netflag

```

func IsUp(v Flags) bool    { return v&FlagUp == FlagUp }
func TurnDown(v *Flags)    { *v &^= FlagUp }
func SetBroadcast(v *Flags) { *v |= FlagBroadcast }
func IsCast(v Flags) bool   { return v&(FlagBroadcast|FlagMulticast) != 0 }
func main() {
    var v Flags = FlagMulticast | FlagUp
    fmt.Printf("%b %t\n", v, IsUp(v))    // "10001 true"
    TurnDown(&v)
    fmt.Printf("%b %t\n", v, IsUp(v))    // "10000 false"
    SetBroadcast(&v)
    fmt.Printf("%b %t\n", v, IsUp(v))    // "10010 false"
    fmt.Printf("%b %t\n", v, IsCast(v))  // "10010 true"
}

```

Como um exemplo mais complexo de **iota**, esta declaração nomeia as potências de 1024:

```

const (
    _ = 1 << (10 * iota)
    KiB // 1024
    MiB // 1048576
    GiB // 1073741824
    TiB // 1099511627776          (excede 1 << 32)
    PiB // 1125899906842624
    EiB // 1152921504606846976
    ZiB // 1180591620717411303424 (excede 1 << 64)
    YiB // 1208925819614629174706176
)

```

O mecanismo de **iota** tem suas limitações. Por exemplo, não é possível gerar as potências mais conhecidas de 1000 (KB, MB e assim por diante) porque não há operador de exponencial.

Exercício 3.13: Escreva declarações **const** para KB, MB, até YB, da forma mais compacta que você puder.

3.6.2 Constantes sem tipo

Constantes em Go são um pouco incomuns. Embora uma constante possa ter qualquer um dos tipos básicos de dados como **int** ou **float64**, incluindo tipos básicos nomeados como **time.Duration**, muitas constantes não estão comprometidas com um tipo em particular. O compilador representa essas constantes sem comprometimento (uncommitted constants) com uma precisão numérica muito maior que valores de tipos básicos, e a aritmética com eles é mais precisa que a aritmética da máquina; você pode pressupor pelo menos 256 bits de precisão. Há seis variantes dessas constantes sem comprometimento, chamadas de booleano *sem tipo* (untyped), inteiro sem tipo, runa sem tipo, número de ponto flutuante sem tipo, número complexo sem tipo e string sem tipo.

Ao adiar esse comprometimento, constantes sem tipo não só preservam sua maior precisão por mais tempo, como também podem participar de muito mais expressões que as constantes comprometidas sem exigir conversões. Por exemplo, os valores **ZiB** e **YiB** do exemplo anterior são grandes demais para armazenar em qualquer variável inteira, mas são constantes legítimas que podem ser usadas em expressões como esta:

```
fmt.Println(YiB/ZiB) // "1024"
```

Como outro exemplo, a constante de ponto flutuante **math.Pi** pode ser usada em qualquer lugar em que um valor de ponto flutuante ou complexo for necessário:

```
var x float32 = math.Pi
var y float64 = math.Pi
var z complex128 = math.Pi
```

Se `math.Pi` estivesse comprometido com um tipo específico, como `float64`, o resultado não seria tão preciso, e conversões de tipo seriam necessárias quando se quisesse usar um valor `float32` ou `complex128`:

```
const Pi64 float64 = math.Pi
var x float32 = float32(Pi64)
var y float64 = Pi64
var z complex128 = complex128(Pi64)
```

Para os literais, a sintaxe determina a variante. Os literais `0`, `0.0`, `0i` e `'\u0000'` representam constantes de mesmo valor, mas são variantes diferentes: inteiro sem tipo, número de ponto flutuante sem tipo, número complexo sem tipo e runa sem tipo, respectivamente. De modo semelhante, `true` e `false` são booleanos sem tipo e strings literais são strings sem tipo.

Lembre-se de que `/` pode representar divisão entre inteiros ou números de ponto flutuante, de acordo com os operandos. Consequentemente, a escolha do literal pode afetar o resultado de uma expressão de divisão de constantes:

```
var f float64 = 212
fmt.Println((f - 32) * 5 / 9)      // "100"; (f - 32) * 5 é um float64
fmt.Println(5 / 9 * (f - 32))     // "0"; 5/9 é um inteiro sem tipo, 0
fmt.Println(5.0 / 9.0 * (f - 32)) // "100"; 5.0/9.0 é um float sem tipo
```

Somente constantes podem ser sem tipo. Quando uma constante sem tipo aparecer do lado direito de uma declaração de variável com um tipo explícito, como na primeira instrução a seguir, ou é atribuída a uma variável, como nas outras três instruções, a constante será implicitamente convertida para o tipo dessa variável, se for possível.

```
var f float64 = 3 + 0i // número complexo sem tipo -> float64
f = 2                // inteiro sem tipo -> float64
f = 1e123             // número de ponto flutuante sem tipo -> float64
f = 'a'              // runa sem tipo -> float64
```

As instruções anteriores, portanto, são equivalentes a:

```
var f float64 = float64(3 + 0i)
f = float64(2)
f = float64(1e123)
```

```
f = float64('a')
```

Seja de modo implícito ou explícito, converter uma constante de um tipo para outro exige que o tipo do alvo possa representar o valor original. Arredondamento é permitido para números de ponto flutuante reais e complexos:

```
const (  
    deadbeef = 0xdeadbeef // int sem tipo com valor 3735928559  
    a = uint32(deadbeef)  // uint32 com valor 3735928559  
    b = float32(deadbeef) // float32 com valor 3735928576 (arredondado para  
cima)  
    c = float64(deadbeef) // float64 com valor 3735928559 (exato)  
    d = int32(deadbeef)   // erro de compilação: constante ultrapassa int32  
    e = float64(1e309)    // compile error: constant overflows float64  
    f = uint(-1)          // erro de compilação: constante é menor que uint  
)
```

Em uma declaração de variável sem um tipo explícito (incluindo declarações curtas de variáveis), a variante da constante sem tipo determina implicitamente o tipo default da variável, como nestes exemplos:

i := 0	// inteiro sem tipo;	int(0) implícito
r := '\000'	// runa sem tipo;	rune('\000') implícito
f := 0.0	// número ponto flutuante sem tipo;	float64(0.0) implícito
c := 0i	// número complexo sem tipo;	complex128(0i) implícito

Observe a assimetria: inteiros sem tipo são convertidos para **int**, cujo tamanho não é garantido, mas números de ponto flutuante e complexos sem tipo são convertidos para os tipos **float64** e **complex128** de tamanhos explícitos. A linguagem não tem tipos **float** e **complex** sem tamanhos análogos ao **int** sem tamanho, pois é muito difícil escrever algoritmos numéricos corretos sem conhecer o tamanho dos tipos de dados de ponto flutuante envolvidos.

Para dar um tipo diferente à variável, devemos converter explicitamente a constante sem tipo para o tipo desejado ou definir o tipo desejado na declaração da variável, como nestes exemplos:

```
var i = int8(0)  
var i int8 = 0
```

Esses defaults são particularmente importantes quando convertemos uma constante sem tipo para um valor de interface (veja o capítulo 7), pois eles determinam seu tipo dinâmico.

```
fmt.Printf("%T\n", 0)           // "int"  
fmt.Printf("%T\n", 0.0)        // "float64"  
fmt.Printf("%T\n", 0i)         // "complex128"  
fmt.Printf("%T\n", '\000')     // "int32" (runa)
```

Já discutimos os tipos de dados básicos de Go. O próximo passo é mostrar como eles podem ser combinados em agrupamentos maiores como arrays e estruturas e, então, em estruturas de dados para solucionar problemas reais de programação; esse é o assunto do capítulo 4.

1 N.T.: O termo *moguls* refere-se aos pequenos montes de neve usados como obstáculos em certas modalidades de esqui.

2 N.T.: Efeito de deixar uma imagem com aspecto quadriculado que ocorre geralmente quando uma imagem de baixa resolução é ampliada.

3 Nota do Revisor Técnico: Esta ordem não é a mais natural para exibir listas ordenadas para pessoas, porque na tabela Unicode, 'Z' < 'a' < 'z' < 'á'. Por este critério, as seguintes palavras estão em ordem lexicográfica: Zorro, alface, zebra, ábaco.

4

Tipos compostos

No capítulo 3 discutimos os tipos básicos, que servem como blocos de construção para estruturas de dados em um programa Go; eles são os átomos de nosso universo. Neste capítulo, daremos uma olhada nos tipos *compostos* – as moléculas criadas pela combinação de tipos básicos de várias maneiras. Discutiremos quatro desses tipos – arrays, fatias (slices), mapas (maps) e estruturas (structs) – e, no final do capítulo, mostraremos como dados estruturados que usam esses tipos podem ser codificados e interpretados como dados JSON e usados para gerar HTML a partir de templates.

Arrays e estruturas são tipos *agregados*; seus valores são concatenações de outros valores em memória. Arrays são homogêneos – todos os seus elementos têm o mesmo tipo – enquanto estruturas são heterogêneas. Tanto arrays quanto estruturas têm tamanho fixo. Por outro lado, fatias e mapas são estruturas de dados dinâmicas que crescem à medida que valores são adicionados.

4.1 Arrays

Um array é uma sequência de tamanho fixo de zero ou mais elementos de um tipo específico. Por causa de seu tamanho fixo, arrays raramente são usados diretamente em Go. As fatias, que podem aumentar e diminuir, são muito mais versáteis, mas para entender as fatias, devemos entender primeiro os arrays.

Elementos individuais de arrays são acessados com a notação convencional de índice, que varia de zero até o tamanho do array menos um. A função embutida `len` devolve o número de elementos do array.

```
var a [3]int           // array de 3 inteiros
```

```

fmt.Println(a[0])           // exibe o primeiro elemento
fmt.Println(a[len(a)-1])   // exibe o último elemento, a[2]
// Exibe os índices e os elementos
for i, v := range a {
    fmt.Printf("%d %d\n", i, v)
}

// Exibe apenas os elementos
for _, v := range a {
    fmt.Printf("%d\n", v)
}

```

Por padrão, os elementos de uma nova variável array inicialmente são definidos com o valor zero do tipo do elemento, que é 0 para números. Podemos usar um *array literal* para inicializar um array com uma lista de valores:

```

var q [3]int = [3]int{1, 2, 3}
var r [3]int = [3]int{1, 2}
fmt.Println(r[2]) // "0"

```

Em um array literal, se reticências “...” estiverem no lugar do tamanho, esse será determinado pela quantidade de inicializadores. A definição de **q** pode ser simplificada assim:

```

q := [...]int{1, 2, 3}
fmt.Printf("%T\n", q) // "[3]int"

```

O tamanho de um array faz parte de seu tipo, portanto **[3]int** e **[4]int** são tipos diferentes. O tamanho deve ser uma expressão constante, ou seja, uma expressão cujo valor pode ser calculado quando o programa é compilado.

```

q := [3]int{1, 2, 3}
q = [4]int{1, 2, 3, 4} // erro de compilação: não é permitido atribuir
                        // [4]int a [3]int

```

Como veremos, a sintaxe literal é semelhante para arrays, fatias, mapas e estruturas. O formato específico anterior apresenta uma lista de valores em sequência, mas também é possível especificar uma lista de pares com índice e valor desta maneira:

```

type Currency int
const (

```

```

    USD Currency = iota
    EUR
    GBP
    RMB
)

symbol := [...]string{USD: "$", EUR: "€", GBP: "£", RMB: "¥"}

fmt.Println(RMB, symbol[RMB]) // "3 ¥"

```

Nesse formato, os índices podem estar em qualquer ordem e alguns podem ser omitidos; como antes, valores não especificados assumem o valor zero do tipo do elemento. Por exemplo:

```
r := [...]int{99: -1}
```

define um array `r` com 100 elementos, todos iguais a zero, exceto o último, cujo valor é -1.

Se o tipo do elemento de um array for *comparável*, o tipo array também será comparável, portanto podemos comparar diretamente dois arrays desse tipo usando o operador `==`, que informa se todos os elementos correspondentes são iguais. O operador `!=` é sua negação.

```

a := [2]int{1, 2}
b := [...]int{1, 2}
c := [2]int{1, 3}
fmt.Println(a == b, a == c, b == c) // "true false false"
d := [3]int{1, 2}
fmt.Println(a == d) // erro de compilação: não é possível
                        // comparar [2]int == [3]int

```

Como um exemplo mais plausível, a função `Sum256` do pacote `crypto/sha256` gera o hash de criptografia SHA256 ou *digest* de uma mensagem armazenada em um slice qualquer de bytes. O digest tem 256 bits, portanto seu tipo é `[32]byte`. Se dois digests forem iguais, é altamente provável que as duas mensagens sejam iguais; se os digests forem diferentes, as duas mensagens serão diferentes. O programa a seguir exibe e compara os digests SHA256 de "x" e "X":


```
import "crypto/sha256"

func main() {
    c1 := sha256.Sum256([]byte("x"))
    c2 := sha256.Sum256([]byte("X"))
    fmt.Printf("%x\n%x\n%t\n%T\n", c1, c2, c1 == c2, c1)
    // Saída:
    // 2d711642b726b04401627ca9fbac32f5c8530fb1903cc4db02258717921a4881
    // 4b68ab3847feda7d6c62c1fbcbeebfa35eab7351ed5e78f4ddadea5df64b8015
    // false
    // [32]uint8
}
```

As duas entradas diferem em apenas um bit, mas aproximadamente metade dos bits são diferentes nos digests. Observe os verbos de **Printf**: **%x** para exibir todos os elementos de um array ou fatia de bytes em hexadecimal, **%t** para mostrar um booleano e **%T** para exibir o tipo de um valor.

Quando uma função é chamada, uma cópia de cada argumento é atribuída à variável de parâmetro correspondente, assim a função recebe uma cópia, não o original. Passar arrays grandes dessa maneira pode ser ineficiente, e qualquer alteração nos elementos do array feita pela função afetará somente a cópia, não o original. Quanto a esse aspecto, Go trata arrays como qualquer outro tipo, mas esse comportamento é diferente de linguagens que passam implicitamente *arrays por referência*.

É claro que podemos passar explicitamente um ponteiro para um array de modo que qualquer modificação nos elementos do array feita pela função seja visível a quem chamou. A função a seguir zera o conteúdo de um array **[32]byte**:

```
func zero(ptr *[32]byte) {
    for i := range ptr {
        ptr[i] = 0
    }
}
```

O array literal **[32]byte{}** gera um array de 32 bytes. Cada elemento do array tem o valor zero para **byte**, que é zero. Podemos usar esse fato para escrever uma versão diferente de **zero**:

```
func zero(ptr *[32]byte) {  
    *ptr = [32]byte{}  
}
```

Usar um ponteiro para um array é eficiente e permite que a função chamada altere a variável de quem chamou, mas os arrays continuam inerentemente inflexíveis por causa de seu tamanho fixo. A função **zero** não aceitará um ponteiro para uma variável **[16]byte**, por exemplo, nem há qualquer maneira de adicionar ou de remover elementos do array. Por esses motivos, além de casos especiais como o hash SHA256 de tamanho fixo, arrays raramente são usados como parâmetros ou resultados de função; em seu lugar, usamos fatias.

Exercício 4.1: Escreva uma função que conte o número de bits diferentes em dois hashes SHA256. (Veja **PopCount** na seção 2.6.2.)

Exercício 4.2: Escreva um programa que exiba o hash SHA256 de sua entrada-padrão por default, mas aceite uma flag de linha de comando para exibir o hash SHA384 ou SHA512 em seu lugar.

4.2 Fatias

Fatias (slices) representam sequências de tamanho variável cujos elementos têm o mesmo tipo. Um tipo fatia é escrito como **[]T**, em que os elementos têm o tipo **T**; ele se parece com um tipo array sem um tamanho.

Arrays e fatias estão intimamente conectados. Uma fatia é uma estrutura de dados leve que dá acesso a uma subsequência de elementos (ou talvez a todos os elementos) de um array conhecido como *array subjacente* da fatia. Uma fatia tem três componentes: um ponteiro, um tamanho (**length**) e uma capacidade (**capacity**). O ponteiro aponta para o primeiro elemento do array acessível pela fatia, que não é necessariamente o primeiro elemento do array. O tamanho é o número de elementos da fatia; ele não pode exceder a capacidade que, normalmente, é o número de elementos entre o início da fatia e o final do array subjacente. As funções embutidas **len** e **cap** devolvem esses valores.

Várias fatias podem compartilhar o mesmo array subjacente e podem referenciar partes desse array que se sobrepõem. A figura 4.1 mostra um array de strings para os meses do ano e duas de suas fatias que se sobrepõem. O array é declarado como:

```
months := [...]string{1: "January", /* ... */, 12: "December"}
```

portanto January é `months[1]` e December é `months[12]`. Normalmente, o elemento do array no índice 0 conteria o primeiro valor, mas como os meses são sempre numerados a partir de 1, podemos excluí-lo da declaração e ele será inicializado com uma string vazia.

O *operador de fatia* `s[i:j]`, em que $0 \leq i \leq j \leq \text{cap}(s)$, cria uma nova fatia que faz referência aos elementos de `i` a `j-1` da sequência `s`, que pode ser uma variável array, um ponteiro para um array ou outra fatia. A fatia resultante tem `j-i` elementos. Se `i` for omitido, ele será igual a 0 e se `j` for omitido, será `len(s)`. Desse modo, a fatia `months[1:13]` refere-se ao intervalo todo de meses válidos, assim como a fatia `months[1:]`; a fatia `months[:]` refere-se ao array todo.

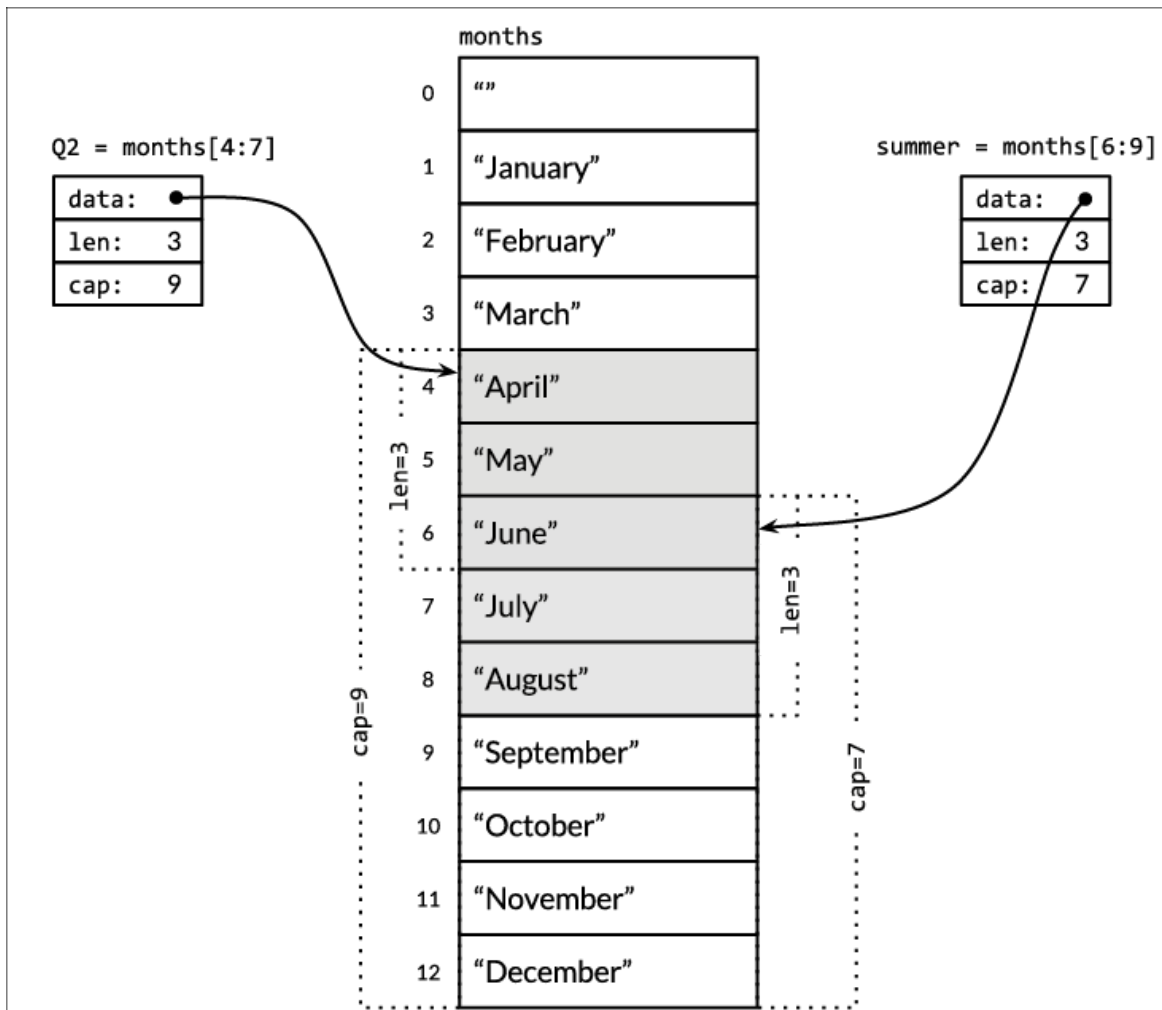


Figura 4.1 – Duas fatias de um array de meses que se sobrepõem.

Vamos definir fatias que se sobrepõem para o segundo trimestre e para o verão no hemisfério norte:

```
Q2 := months[4:7]
summer := months[6:9]
fmt.Println(Q2)      // ["April" "May" "June"]
fmt.Println(summer)  // ["June" "July" "August"]
```

“June” está incluído nas duas fatias e é a única saída do (ineficiente) teste a seguir, que exhibe os elementos comuns:

```
for _, s := range summer {
    for _, q := range Q2 {
        if s == q {
            fmt.Printf("%s appears in both\n", s)
        }
    }
}
```

```

    }
}

```

Fatiar além de **cap(s)** provoca pânico, mas além de **len(s)** estende a fatia, portanto o resultado pode ser maior que o original:

```

fmt.Println(summer[:20])    // pânico: fora do intervalo
endlessSummer := summer[:5] // estende uma fatia (dentro da capacidade)
fmt.Println(endlessSummer)  // "[June July August September October]"

```

Como informação adicional, observe a semelhança entre a operação de substring em strings e o operador de fatia em fatias **[]byte**. Ambos são escritos como **x[m:n]** e ambos devolvem uma subsequência dos bytes originais, compartilhando a representação subjacente e, sendo assim, ambas as operações consomem o mesmo tempo. A expressão **x[m:n]** produz uma string se **x** for uma string, ou um **[]byte** se **x** for um **[]byte**.

Como uma fatia contém um ponteiro para um elemento de um array, passar uma fatia a uma função permite a essa função modificar os elementos do array subjacente. Em outras palavras, copiar uma fatia cria um *apelido* ou *alias* (seção 2.3.2) para o array subjacente. A função **reverse** inverte os elementos de uma fatia **[]int** in-place, e pode ser aplicada a fatias de qualquer tamanho.

gopl.io/ch4/rev

```

// reverse inverte uma fatia de ints in-place
func reverse(s []int) {
    for i, j := 0, len(s)-1; i < j; i, j = i+1, j-1 {
        s[i], s[j] = s[j], s[i]
    }
}

```

Neste caso, invertemos todo o array **a**:

```

a := [...]int{0, 1, 2, 3, 4, 5}
reverse(a[:])
fmt.Println(a) // "[5 4 3 2 1 0]"

```

Uma forma simples de fazer uma *rotação* (rotate) à esquerda de uma fatia de *n* elementos é aplicar a função **reverse** três vezes, inicialmente nos primeiros *n* elementos, em seguida nos elementos restantes e, por fim, na fatia toda. (Para uma rotação à direita, faça a terceira chamada antes.)

```
s := []int{0, 1, 2, 3, 4, 5}
// Faz a rotação de s à esquerda em duas posições
reverse(s[:2])
reverse(s[2:])
reverse(s)
fmt.Println(s) // "[2 3 4 5 0 1]"
```

Observe como a expressão que inicializa a fatia **s** difere daquela usada no array **a**. Uma *fatia literal* se parece com um array literal, ou seja, uma sequência de valores separados por vírgulas e entre chaves, mas o tamanho não é especificado. Isso cria implicitamente uma variável do tipo array do tamanho correto e produz uma fatia que aponta para ele. Como ocorre com arrays literais, as fatias literais podem especificar valores em sequência, definir seus índices explicitamente ou usar uma combinação dos dois estilos.

De modo diferente dos arrays, as fatias não são comparáveis, portanto não podemos usar `==` para testar se duas fatias contêm os mesmos elementos. A biblioteca-padrão oferece a função **bytes.Equal**, extremamente otimizada, para comparar duas fatias de bytes (`[]byte`), mas para outros tipos de fatia, devemos implementar a comparação por conta própria:

```
func equal(x, y []string) bool {
    if len(x) != len(y) {
        return false
    }
    for i := range x {
        if x[i] != y[i] {
            return false
        }
    }
    return true
}
```

Dado o modo como esse teste “profundo” de igualdade é natural e não é mais custoso em tempo de execução que o operador `==` para arrays de strings, pode ser intrigante que comparações entre fatias não funcionem também dessa maneira. Há dois motivos pelos quais a equivalência profunda é problemática. Em primeiro lugar, diferentemente dos elementos de um array, os elementos de uma fatia são indiretos, fazendo

com que seja possível que uma fatia contenha a si mesma. Embora haja maneiras de lidar com casos desse tipo, nenhuma delas é simples, eficiente e, acima de tudo, óbvia.

Em segundo lugar, como os elementos de uma fatia são indiretos, o valor de uma fatia fixa pode conter diferentes elementos em instantes diferentes à medida que o conteúdo do array subjacente é modificado. Pelo fato de uma tabela hash como o tipo mapa de Go fazer apenas cópias rasas (shallow copies) de suas chaves, é necessário que a igualdade para cada chave permaneça a mesma durante o tempo de vida da tabela hash. Assim a equivalência profunda deixaria as fatias inadequadas para uso como chaves de mapa. Para tipos referência, como ponteiros e canais, o operador `==` testa a *identidade de referência*, isto é, se duas entidades se referem ao mesmo dado.

Um teste de igualdade “raso” análogo para fatias poderia ser útil e resolveria o problema com mapas, mas o tratamento inconsistente entre fatias e arrays pelo operador `==` seria confuso. A opção mais segura é não permitir comparações entre fatias.

A única comparação permitida entre fatias é com `nil`, como em:

```
if summer == nil { /* ... */ }
```

O valor zero de um tipo fatia é `nil`. Uma fatia nil não tem array subjacente. A fatia nil tem tamanho e capacidade iguais a zero, mas há também fatias que não são nil de tamanho e capacidade iguais a zero, como `[]int{}` ou `make([]int, 3)[3:]`. Como ocorre com qualquer tipo que possa ter valores nil, o valor nil de um tipo particular de fatia pode ser escrito usando uma expressão de conversão como `[]int(nil)`.

```
var s []int    // len(s) == 0, s == nil
s = nil        // len(s) == 0, s == nil
s = []int(nil) // len(s) == 0, s == nil
s = []int{}     // len(s) == 0, s != nil
```

Portanto, se você precisa testar se uma fatia está vazia, use `len(s) == 0`, e não `s == nil`. Além de ser igual a `nil`, uma fatia nil comporta-se como qualquer outra fatia de tamanho zero; por exemplo, `reverse(nil)` é totalmente seguro. A menos que esteja claramente documentado de forma

diferente, funções Go devem tratar todas as fatias de tamanho zero da mesma maneira, sejam elas nil ou não.

A função embutida **make** cria uma fatia de um tipo de elemento, um tamanho e uma capacidade especificados. O argumento para capacidade pode ser omitido, caso em que ele será igual ao tamanho.

```
make([]T, len)
make([]T, len, cap) // mesmo que make([]T, cap)[:len]
```

Internamente, **make** cria uma variável do tipo array sem nome e devolve uma fatia dela; o array é acessível somente por meio da fatia devolvida. No primeiro formato, a fatia é uma visão de todo o array. No segundo, a fatia é uma visão apenas dos primeiros **len** elementos do array, mas sua capacidade inclui todo o array. Os elementos adicionais são reservados para crescimento futuro.

4.2.1 Função **append**

A função embutida **append** concatena um item a uma fatia:

```
var runes []rune
for _, r := range "Hello, 世界" {
    runes = append(runes, r)
}
fmt.Printf("%q\n", runes) // "['H' 'e' 'l' 'l' 'o' ' ',' ' ' '世' '界']"
```

O laço usa **append** para criar a fatia de nove runas codificadas pela string literal, embora esse problema específico seja mais convenientemente resolvido com o uso da conversão embutida `[]rune("Hello, 世界")`.

A função **append** é fundamental para entender o funcionamento das fatias, portanto vamos dar uma olhada no que está acontecendo. Eis uma versão chamada **appendInt**, especializada para fatias `[]int`:

gopl.io/ch4/append

```
func appendInt(x []int, y int) []int {
    var z []int
    zlen := len(x) + 1
    if zlen <= cap(x) {
        // Há espaço para crescer. Estende a fatia
```



```

    z = x[:zlen]
} else {
    // Não há espaço suficiente. Aloca um novo array
    // Cresce para o dobro, para complexidade linear amortizada
    zcap := zlen
    if zcap < 2*len(x) {
        zcap = 2 * len(x)
    }
    z = make([]int, zlen, zcap)
    copy(z, x)    // uma função embutida; veja o texto
}
z[len(x)] = y
return z
}

```

Toda chamada a **appendInt** deve verificar se a fatia tem capacidade suficiente para armazenar os novos elementos no array existente. Em caso afirmativo, a fatia é estendida definindo-se uma fatia maior (ainda dentro do array original), o elemento **y** é copiado para o novo espaço e a fatia é devolvida. A entrada **x** e o resultado **z** compartilham o mesmo array subjacente.

Se não houver espaço suficiente para crescimento, **appendInt** deve alocar um novo array grande o suficiente para armazenar o resultado, copiar os valores de **x** para ele e então concatenar o novo elemento **y**. O resultado **z** agora referencia um array subjacente diferente do array referenciado por **x**.

Seria simples copiar os elementos com laços explícitos, mas é mais fácil usar a função embutida **copy**, que copia elementos de uma fatia para outra do mesmo tipo. Seu primeiro argumento é o destino e o segundo é a origem, que lembra a ordem dos operandos em uma atribuição como **dst = src**. As fatias podem referenciar o mesmo array subjacente; elas podem até mesmo se sobrepor. Embora não seja usado aqui, **copy** devolve o número de elementos copiados, que é o menor número entre os tamanhos das duas fatias, portanto não há perigo de ultrapassar o final nem de escrever algo fora do intervalo.

Por questões de eficiência, o novo array normalmente é maior que o

mínimo necessário para armazenar **x** e **y**. Expandir o array dobrando seu tamanho a cada expansão evita uma quantidade excessiva de alocações e garante que concatenar um único elemento, em média, demore o mesmo tempo. O programa a seguir mostra o efeito:

```
func main() {  
    var x, y []int  
    for i := 0; i < 10; i++ {  
        y = appendInt(x, i)  
        fmt.Printf("%d cap=%d\t%v\n", i, cap(y), y)  
        x = y  
    }  
}
```

Cada mudança de capacidade indica uma alocação e uma cópia:

```
0 cap=1  [0]  
1 cap=2  [0 1]  
2 cap=4  [0 1 2]  
3 cap=4  [0 1 2 3]  
4 cap=8  [0 1 2 3 4]  
5 cap=8  [0 1 2 3 4 5]  
6 cap=8  [0 1 2 3 4 5 6]  
7 cap=8  [0 1 2 3 4 5 6 7]  
8 cap=16 [0 1 2 3 4 5 6 7 8]  
9 cap=16 [0 1 2 3 4 5 6 7 8 9]
```

Vamos dar uma olhada mais de perto na iteração **i=3**. A fatia **x** contém os três elementos **[0 1 2]**, mas tem capacidade **4**, portanto há um único elemento extra no final e **appendInt** para o elemento **3** pode continuar sem realocação. A fatia **y** resultante tem tamanho e capacidade iguais a **4** e tem o mesmo array subjacente da fatia original **x**, como mostra a figura 4.2.

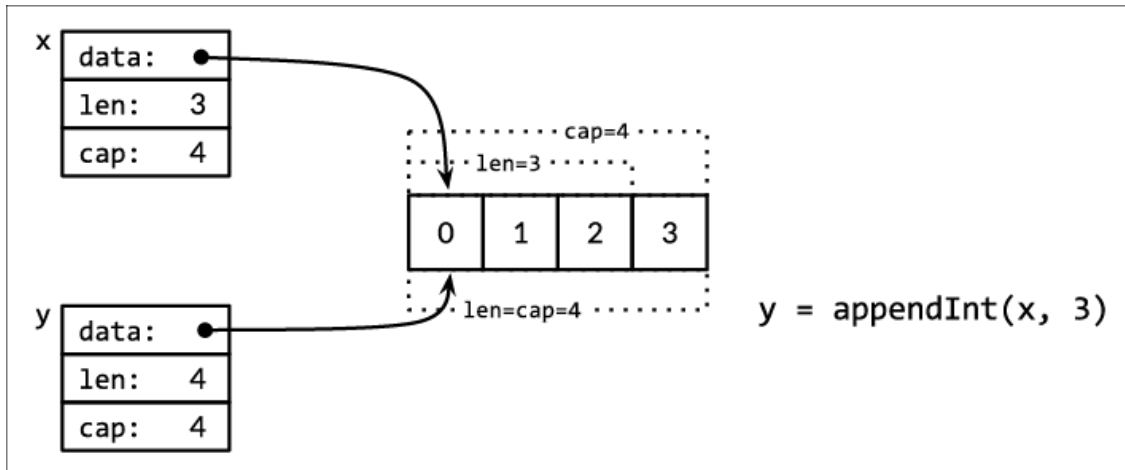


Figura 4.2 – Concatenando com espaço para crescer.

Na próxima iteração, $i=4$, não há nenhuma folga, portanto `appendInt` aloca um novo array de tamanho 8, copia os quatro elementos [0 1 2 3] de **x** e concatena 4, que é o valor de *i*. A fatia resultante **y** tem tamanho 5, mas a capacidade é igual a 8; a folga de 3 fará com que as três próximas iterações não precisem de realocação. As fatias **y** e **x** são visões de arrays diferentes. Essa operação está representada na figura 4.3.

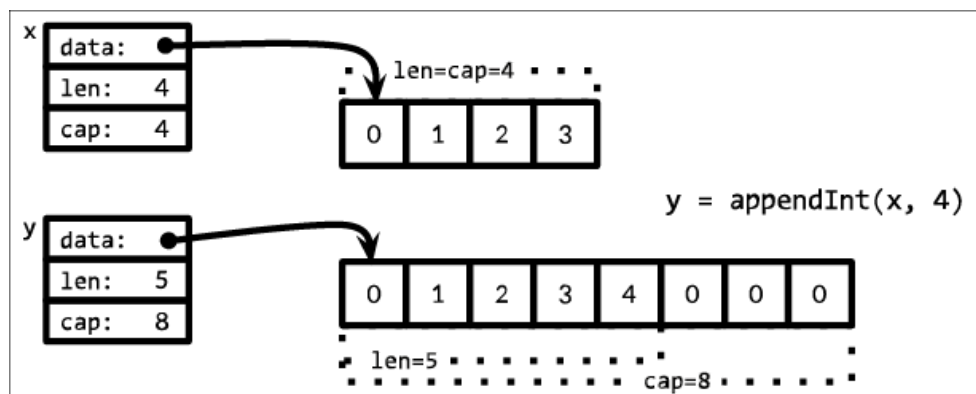


Figura 4.3 – Concatenando sem espaço para crescer.

A função embutida `append` pode usar uma estratégia de crescimento mais sofisticada que a função simplista `appendInt`. Normalmente, não sabemos se uma dada chamada a `append` provocará uma realocação, desta forma não podemos supor que a fatia original se refira ao mesmo array que a fatia resultante, nem se ela se refere a um array diferente. De modo semelhante, não devemos supor que atribuições a elementos da fatia antiga se refletirão (ou não) na nova fatia. Consequentemente, é comum

atribuir o resultado de uma chamada a **append** à mesma variável de fatia cujo valor passamos para **append**:

```
runes = append(runes, r)
```

Atualizar a variável de fatia é necessário não só quando chamamos **append**, mas para qualquer função que possa alterar o tamanho ou a capacidade de uma fatia ou fazê-la referenciar um array subjacente diferente. Para usar fatias corretamente é importante ter em mente que, embora os elementos do array subjacente sejam indiretos, o ponteiro da fatia, o tamanho e a capacidade não o são. Atualizá-los exige uma atribuição como a que vimos anteriormente. Nesse aspecto, as fatias não são tipos referência “puros”, mas lembram um tipo agregado como a estrutura a seguir:

```
type IntSlice struct {  
    ptr      *int  
    len, cap int  
}
```

Nossa função **appendInt** adiciona um único elemento a uma fatia, mas a função embutida **append** nos permite acrescentar mais de um novo elemento, ou até mesmo uma fatia inteira.

```
var x []int  
x = append(x, 1)  
x = append(x, 2, 3)  
x = append(x, 4, 5, 6)  
x = append(x, x...) // concatena a fatia x  
fmt.Println(x)      // "[1 2 3 4 5 6 1 2 3 4 5 6]"
```

Com a pequena modificação mostrada a seguir, podemos chegar ao comportamento da função embutida **append**. As reticências “...” na declaração de **appendInt** deixam a função *variádica*: ela aceita qualquer número de argumentos finais. As reticências correspondentes na chamada anterior a **append** mostram como fornecer uma lista de argumentos a partir de uma fatia. Explicaremos esse mecanismo em detalhes na seção 5.7.

```
func appendInt(x []int, y ...int) []int {  
    var z []int
```

```

    zlen := len(x) + len(y)
    // ...expande z para no mínimo zlen...
    copy(z[len(x):], y)
    return z
}

```

A lógica para expandir o array subjacente de **z** permanece inalterada e não foi mostrada.

4.2.2 Técnicas in-place para fatias

Vamos ver mais exemplos de funções que, como **rotate** e **reverse**, modificam os elementos de uma fatia in-place. Dada uma lista de strings, a função **nonempty** devolve as strings não vazias:

gopl.io/ch4/nonempty

```

// Nonempty é um exemplo de um algoritmo in-place para fatias
package main

import "fmt"

// nonempty devolve uma fatia que armazena apenas as strings não vazias
// O array subjacente é modificado durante a chamada
func nonempty(strings []string) []string {
    i := 0
    for _, s := range strings {
        if s != "" {
            strings[i] = s
            i++
        }
    }
    return strings[:i]
}

```

A parte sutil é que a fatia de entrada e a fatia de saída compartilham o mesmo array subjacente. Isso evita a necessidade de alocar outro array, embora, é claro, o conteúdo de **data** seja parcialmente sobrescrito, como evidenciado pela segunda instrução de exibição:

```

data := []string{"one", "", "three"}
fmt.Printf("%q\n", nonempty(data)) // `["one" "three"]`
fmt.Printf("%q\n", data)           // `["one" "three" "three"]`

```

Assim, normalmente escreveríamos: **data = nonempty(data)**.

A função **nonempty** também pode ser escrita usando **append**:

```
func nonempty2(strings []string) []string {
    out := strings[:0] // fatia de tamanho zero do original
    for _, s := range strings {
        if s != "" {
            out = append(out, s)
        }
    }
    return out
}
```

Qualquer que seja a variante usada, reutilizar um array dessa maneira exige que, no máximo, um valor de saída seja produzido para cada valor de entrada, o que é verdadeiro para muitos algoritmos que filtram elementos de uma sequência ou combinam elementos adjacentes. Um uso intrincado de fatias como esse não é a regra, mas a exceção, mas pode ser claro, eficiente e útil ocasionalmente.

Uma fatia pode ser usada para implementar uma pilha (stack). Dada uma fatia **stack** inicialmente vazia, podemos inserir um novo valor (push) no final da fatia usando **append**:

```
stack = append(stack, v) // push de v
```

O topo da pilha é o último elemento:

```
top := stack[len(stack)-1] // topo da pilha
```

e diminuir a pilha removendo esse elemento é feito assim:

```
stack = stack[:len(stack)-1] // pop
```

Para remover um elemento do meio de uma fatia, preservando a ordem dos elementos remanescentes, use **copy** para deslocar os elementos de posições mais altas em um para preencher a lacuna:

```
func remove(slice []int, i int) []int {
    copy(slice[i:], slice[i+1:])
    return slice[:len(slice)-1]
}

func main() {
    s := []int{5, 6, 7, 8, 9}
    fmt.Println(remove(s, 2)) // "[5 6 8 9]"
}
```

Se não for necessário preservar a ordem, podemos simplesmente mover o último elemento para a lacuna:

```
func remove(slice []int, i int) []int {
    slice[i] = slice[len(slice)-1]
    return slice[:len(slice)-1]
}

func main() {
    s := []int{5, 6, 7, 8, 9}
    fmt.Println(remove(s, 2))    // "[5 6 9 8]"
}
```

Exercício 4.3: Reescreva **reverse** usando um ponteiro de array no lugar de uma fatia.

Exercício 4.4: Escreva uma versão de **rotate** que funcione com um único passo.

Exercício 4.5: Escreva uma função in-place para eliminar duplicatas adjacentes em uma fatia `[]string`.

Exercício 4.6: Escreva uma função in-place que transforme toda sequência de espaços Unicode adjacentes (veja `unicode.IsSpace`) de uma fatia `[]byte` codificada em UTF-8 em um único espaço ASCII.

Exercício 4.7: Modifique **reverse** para inverter os caracteres de uma fatia `[]byte` que representa uma string codificada em UTF-8, in-place. Você é capaz de fazer isso sem alocar uma nova memória?

4.3 Mapas

A *hash table* (tabela de dispersão ou tabela hash) é uma das estruturas de dados mais engenhosas e versáteis. É uma coleção não ordenada de pares chave/valor em que cada a chave é única e o valor associado a uma dada chave pode ser recuperado, atualizado ou removido usando, em média, um número constante de comparações de chaves, independentemente do tamanho da tabela.

Em Go, um *mapa* (map) é uma referência a uma tabela hash, e um tipo mapa é escrito como `map[K]V`, em que **K** e **V** são os tipos das chaves e dos

valores. Todas as chaves em um dado mapa são do mesmo tipo, e todos os valores são do mesmo tipo, mas as chaves e os valores não precisam ser do mesmo tipo. O tipo de chave **K** deve ser comparável com `==`, de modo que o mapa possa testar se uma dada chave é igual a uma que ele já contém. Embora números de ponto flutuante sejam comparáveis, compará-los para ver se são iguais não é uma boa ideia e, como mencionamos no capítulo 3, especialmente ruim se NaN for um valor possível. Não há restrições para o tipo **V** dos valores.

A função embutida **make** pode ser usada para criar um mapa:

```
ages := make(map[string]int) // mapeamento de strings para ints
```

Também podemos usar um *mapa literal* para criar um novo mapa preenchido com alguns pares chave/valor iniciais:

```
ages := map[string]int{
    "alice": 31,
    "charlie": 34,
}
```

Isso é equivalente a:

```
ages := make(map[string]int)
ages["alice"] = 31
ages["charlie"] = 34
```

portanto uma expressão alternativa para um novo mapa vazio é `map[string]int{}`.

Elementos de mapa são acessados por meio da notação usual de indexação:

```
ages["alice"] = 32
fmt.Println(ages["alice"]) // "32"
```

e podem ser removidos com a função embutida **delete**:

```
delete(ages, "alice") // remove o elemento ages["alice"]
```

Todas essas operações são seguras, mesmo se o elemento não estiver no mapa; uma busca no mapa usando uma chave que não está presente devolve o valor zero para seu tipo; por exemplo, o código a seguir funciona, mesmo quando **"bob"** ainda não é uma chave do mapa porque o valor de `ages["bob"]` será 0.


```
ages["bob"] = ages["bob"] + 1 // feliz aniversário!
```

As formas abreviadas de atribuição `x += y` e `x++` também funcionam para elementos de mapa, portanto podemos reescrever a instrução anterior assim:

```
ages["bob"] += 1
```

ou, de modo mais compacto ainda:

```
ages["bob"]++
```

Porém, um elemento de mapa não é uma variável, e não podemos obter seu endereço:

```
_ = &ages["bob"] // erro de compilação: não é possível obter o endereço
                  // de um elemento de mapa
```

Um dos motivos pelos quais não podemos obter o endereço de um elemento de mapa é que aumentar um mapa pode alterar o hashing de elementos existentes que serão salvos em novos locais de armazenamento, invalidando potencialmente o endereço.

Para enumerar todos os pares chave/valor do mapa, usamos um loop **for** baseado em **range**, semelhante àqueles que vimos para fatias. Iterações sucessivas do loop fazem as variáveis **name** e **age** serem definidas com o próximo par chave/valor:

```
for name, age := range ages {
    fmt.Printf("%s\t%d\n", name, age)
}
```

A ordem da iteração no mapa não é especificada, e diferentes implementações podem usar uma função diferente de hash, resultando em uma ordem diferente. Na prática, a ordem é aleatória, variando de uma execução para outra. Isso é proposital: fazer a sequência variar ajuda a forçar os programas a serem robustos entre diferentes implementações. Para listar os pares chave/valor em ordem, devemos ordenar as chaves explicitamente, por exemplo, usando a função **Strings** do pacote **sort** se as chaves forem strings. Este é um padrão comum:

```
import "sort"

var names []string
for name := range ages {
```

```

    names = append(names, name)
}
sort.Strings(names)
for _, name := range names {
    fmt.Printf("%s\t%d\n", name, ages[name])
}

```

Como conhecemos o tamanho final de **names** desde o início, é mais eficaz alocar um array do tamanho necessário previamente. A instrução a seguir cria uma fatia que, inicialmente, está vazia, mas tem capacidade suficiente para armazenar todas as chaves do mapa **ages**:

```
names := make([]string, 0, len(ages))
```

No primeiro loop **range** anterior, precisamos somente das chaves do mapa **ages**, portanto omitimos a segunda variável do loop. No segundo loop, precisamos apenas dos elementos da fatia **names**, desta forma usamos o identificador vazio **_** para ignorar a primeira variável, que é o índice.

O valor zero de um tipo mapa é **nil**, ou seja, uma referência para nenhuma tabela hash.

```

var ages map[string]int
fmt.Println(ages == nil)    // "true"
fmt.Println(len(ages) == 0) // "true"

```

A maioria das operações com mapas, incluindo buscas, **delete**, **len** e loops **range**, é executada de forma segura em uma referência **nil** de mapa, pois ela se comporta como um mapa vazio. Porém, fazer um armazenamento em um mapa **nil** causa pânico:

```
ages["carol"] = 21 // pânico: atribuição a uma entrada em um mapa nil
```

Você deve alocar o mapa antes de poder armazenar valores nele.

Acessar um elemento do mapa por meio de indexação sempre produz um valor. Se a chave está presente no mapa, você obterá o valor correspondente; caso contrário, o valor zero para o tipo do elemento será obtido, como vimos com **ages["bob"]**. Em muitos casos, não há problemas, porém, às vezes, você precisará saber se o elemento estava presente ou não. Por exemplo, se o tipo do elemento for numérico, talvez

você precise distinguir entre um elemento inexistente e um elemento que, por acaso, tenha o valor zero, usando um teste como este:

```
age, ok := ages["bob"]
if !ok { /* "bob" não é uma chave nesse mapa; age == 0. */ }
```

Com frequência, você verá essas duas instruções combinadas assim:

```
if age, ok := ages["bob"]; !ok { /* ... */ }
```

Usar indexação em um mapa nesse contexto produz dois valores; o segundo valor é um booleano que informa se o elemento estava presente. A variável booleana geralmente é chamada de **ok**, em especial se ela for usada imediatamente em uma condição **if**.

Como ocorre com as fatias, os mapas não podem ser comparados uns com os outros; a única comparação permitida é com **nil**. Para testar se dois mapas contêm as mesmas chaves e os mesmos valores associados, devemos implementar um loop:

```
func equal(x, y map[string]int) bool {
    if len(x) != len(y) {
        return false
    }
    for k, xv := range x {
        if yv, ok := y[k]; !ok || yv != xv {
            return false
        }
    }
    return true
}
```

Observe como usamos **!ok** para distinguir os casos “ausente” e “presente, mas igual a zero”. Se tivéssemos ingenuamente escrito **xv != y[k]**, a chamada a seguir informaria incorretamente que seus argumentos são iguais:

```
// Verdadeiro se equal for implementado incorretamente
equal(map[string]int{"A": 0}, map[string]int{"B": 42})
```

A linguagem Go não tem um tipo **set**, mas como as chaves de um mapa são distintos, um mapa pode servir a esse propósito. Para ilustrar, o programa **dedup** lê uma sequência de linhas e exibe apenas a primeira

ocorrência de cada linha distinta. (É uma variante do programa **dup** que mostramos na seção 1.3.) O programa **dedup** usa um mapa cujas chaves representam o conjunto de linhas que já apareceram para garantir que ocorrências subsequentes não sejam exibidas.

gopl.io/ch4/dedup

```
func main() {
    seen := make(map[string]bool) // um conjunto de strings
    input := bufio.NewScanner(os.Stdin)
    for input.Scan() {
        line := input.Text()
        if !seen[line] {
            seen[line] = true
            fmt.Println(line)
        }
    }
    if err := input.Err(); err != nil {
        fmt.Fprintf(os.Stderr, "dedup: %v\n", err)
        os.Exit(1)
    }
}
```

Programadores Go muitas vezes descrevem um mapa usado dessa maneira como um “conjunto de strings”, sem maiores preocupações, mas tome cuidado, pois nem todos os valores `map[string]bool` são realmente conjuntos; alguns podem conter valores **true** e **false**.

Às vezes, precisamos de um mapa ou de um conjunto cujas chaves sejam fatias, mas pelo fato de as chaves de um mapa terem de ser comparáveis, isso não pode ser expresso diretamente. No entanto, pode ser feito em dois passos. Inicialmente, definimos uma função auxiliar *k* que mapeia cada chave a uma string, com a propriedade $k(x) == k(y)$ se e somente se consideramos *x* e *y* equivalentes. Em seguida, criamos um mapa cujas chaves sejam strings, aplicando a função auxiliar a cada chave antes de acessarmos o mapa.

O exemplo a seguir usa um mapa para registrar o número de vezes que **Add** foi chamado com uma dada lista de strings. Ele usa `fmt.Sprintf` para converter uma fatia de strings em uma única string que seja

apropriada como chave de mapa, usando aspas em cada elemento da fatia com %q para registrar fielmente as fronteiras das strings:

```
var m = make(map[string]int)

func k(list []string) string { return fmt.Sprintf("%q", list) }

func Add(list []string) { m[k(list)]++ }

func Count(list []string) int { return m[k(list)] }
```

A mesma abordagem pode ser usada para qualquer tipo não comparável de chave, e não apenas para fatias. Essa abordagem é útil até mesmo para tipos comparáveis de chaves quando você quiser uma definição de igualdade que não seja ==, como comparações de strings sem diferenciar letras maiúsculas de minúsculas. Além disso, o tipo de k(x) não precisa ser uma string; qualquer tipo comparável com a propriedade desejada de equivalência servirá, por exemplo, inteiros, arrays ou estruturas.

A seguir, apresentamos outro exemplo de mapas em ação: um programa que conta as ocorrências de cada ponto de código Unicode distinto em sua entrada. Como há um grande número de caracteres possíveis e somente uma pequena fração deles apareceria em qualquer documento em particular, um mapa é uma forma natural de registrar apenas aqueles que forem vistos e seus contadores correspondentes.

gopl.io/ch4/charcount

```
// Charcount conta caracteres Unicode
package main

import (
    "bufio"
    "fmt"
    "io"
    "os"
    "unicode"
    "unicode/utf8"
)

func main() {
    counts := make(map[rune]int) // contagens dos caracteres Unicode
    var utflen [utf8.UTFMax + 1]int // contagem de tamanhos das
                                   // codificações UTF-8
    invalid := 0 // contagem de caracteres UTF-8 inválidos
```

```

in := bufio.NewReader(os.Stdin)
for {
    r, n, err := in.ReadRune() // retorna runa, número de bytes, erro
    if err == io.EOF {
        break
    }
    if err != nil {
        fmt.Fprintf(os.Stderr, "charcount: %v\n", err)
        os.Exit(1)
    }
    if r == unicode.ReplacementChar && n == 1 {
        invalid++
        continue
    }
    counts[r]++
    utflen[n]++
}
fmt.Printf("rune\tcount\n")
for c, n := range counts {
    fmt.Printf("%q\t%d\n", c, n)
}
fmt.Print("\nlen\tcount\n")
for i, n := range utflen {
    if i > 0 {
        fmt.Printf("%d\t%d\n", i, n)
    }
}
if invalid > 0 {
    fmt.Printf("\n%d invalid UTF-8 characters\n", invalid)
}
}

```

O método **ReadRune** faz a decodificação de UTF-8 e devolve três valores: a runa decodificada, o tamanho em bytes de sua codificação UTF-8 e um valor de erro. O único erro esperado é o fim de arquivo (end-of-file). Se a entrada não for uma codificação UTF-8 permitida para uma runa, a runa devolvida será **unicode.ReplacementChar** e o tamanho será 1.

O programa **charcount** também exibe um contador dos tamanhos das codificações UTF-8 das runas que aparecem na entrada. Um mapa não é a melhor estrutura de dados para isso; como os tamanhos da codificação variam somente de 1 a **utf8.UTFMax** (cujo valor é 4), um array é mais

compacto.

Como experimento, executamos **charcount** na versão original deste livro em determinado ponto. Embora a maior parte dele esteja em inglês, é claro, ele tem um bom número de caracteres que não são ASCII. Eis os dez mais frequentes:

° 27 世 15 界 14 é 13 × 10 ≤ 5 × 5 国 4 (4 □ 3

e aqui está a distribuição dos tamanhos de todas as codificações UTF-8:

len	count
1	765391
2	60
3	70
4	0

O tipo do valor de um mapa pode, ele próprio, ser um tipo composto, como um mapa ou uma fatia. No código a seguir, o tipo da chave de **graph** é **string** e o tipo do valor é **map[string]bool**, que representa um conjunto de strings. Conceitualmente, **graph** mapeia uma string a um conjunto de strings relacionadas, seus sucessores em um grafo dirigido (*directed graph*).

gopl.io/ch4/graph

```
var graph = make(map[string]map[string]bool)

func addEdge(from, to string) {
    edges := graph[from]
    if edges == nil {
        edges = make(map[string]bool)
        graph[from] = edges
    }
    edges[to] = true
}

func hasEdge(from, to string) bool {
    return graph[from][to]
}
```

A função **addEdge** mostra a maneira idiomática de preencher um mapa em modo lazy (preguiçoso), isto é, inicializar cada valor à medida que sua chave aparecer pela primeira vez. A função **hasEdge** mostra como o valor zero de uma entrada ausente no mapa muitas vezes é útil: mesmo que

nem **from** nem **to** estejam presentes, **graph[from][to]** sempre fornecerá um resultado que faz sentido.

Exercício 4.8: Modifique **charcount** para contar letras, dígitos e assim por diante de acordo com suas categorias Unicode, usando funções como **unicode.IsLetter**.

Exercício 4.9: Escreva um programa **wordfreq** para informar a frequência de cada palavra em um arquivo-texto de entrada. Chame **input.Split(bufio.ScanWords)** antes da primeira chamada a **Scan** para separar a entrada em palavras, e não em linhas.

4.4 Estruturas

Uma *estrutura* é um tipo de dado agregado que agrupa zero ou mais valores nomeados de tipos quaisquer como uma única entidade. Cada valor é chamado de *campo*. O exemplo clássico de uma estrutura para processamento de dados é o registro de funcionário, cujos campos são um ID único, o nome do funcionário, endereço, data de nascimento, cargo, salário, gerente e informações afins. Todos esses campos são reunidos em uma só entidade que pode ser copiada como uma unidade, passada para funções e devolvida por elas, armazenada em arrays e assim por diante.

As duas instruções a seguir declaram uma estrutura chamada **Employee** e uma variável chamada **dilbert** que é uma instância de **Employee**:

```
type Employee struct {
    ID          int
    Name        string
    Address      string
    DoB         time.Time
    Position    string
    Salary      int
    ManagerID   int
}

var dilbert Employee
```

Os campos individuais de **dilbert** são acessados com a notação de ponto, por exemplo, **dilbert.Name** e **dilbert.DoB**. Como **dilbert** é uma

variável, seus campos também são variáveis, portanto podemos fazer atribuições a um campo:

```
dilbert.Salary -= 5000 // reduzido por escrever poucas linhas de código
```

ou tomar seu endereço e acessá-lo por meio de um ponteiro:

```
position := &dilbert.Position
*position = "Senior " + *position    // promovido, por terceirizar funções
                                     // para a Elbônia
```

A notação de ponto também funciona com um ponteiro para uma estrutura:

```
var employeeOfTheMonth *Employee = &dilbert
employeeOfTheMonth.Position += " (proactive team player)"
```

A última instrução é equivalente a:

```
(*employeeOfTheMonth).Position += " (proactive team player)"
```

Dado o ID único de um funcionário, a função **EmployeeByID** devolve um ponteiro para uma estrutura **Employee**. Podemos usar a notação de ponto para acessar seus campos:

```
func EmployeeByID(id int) *Employee { /* ... */ }

fmt.Println(EmployeeByID(dilbert.ManagerID).Position) // "Pointy-haired boss"

id := dilbert.ID
EmployeeByID(id).Salary = 0 // despedido... sem um verdadeiro motivo
```

A última instrução atualiza a estrutura **Employee** apontada pelo resultado da chamada a **EmployeeByID**. Se o tipo de resultado de **EmployeeByID** fosse alterado para **Employee** em vez de ***Employee**, a instrução de atribuição não compilaria, pois seu lado esquerdo não identificaria uma variável.

Campos normalmente são escritos um por linha, com o seu nome antes do tipo, no entanto campos consecutivos de mesmo tipo podem ser combinados, como no caso de **Name** e **Address** a seguir:

```
type Employee struct {
    ID          int
    Name, Address string
    DoB         time.Time
    Position    string
}
```

```

    Salary      int
    ManagerID   int
}

```

A ordem dos campos é significativa para a identidade do tipo. Se tivéssemos combinado também a declaração do campo **Position** (também é uma string) ou trocado a ordem de **Name** e **Address**, estaríamos definindo um tipo diferente de estrutura. Normalmente, só combinamos as declarações de campos relacionados.

O nome do campo de uma estrutura é exportado se começar com uma letra maiúscula; esse é o sistema principal de controle de acesso de Go. Um tipo estrutura pode conter uma mistura de campos exportados e não exportados.

Os tipos estrutura tendem a ser mais extenso porque frequentemente envolvem uma linha para cada campo. Embora possamos escrever o tipo todo sempre que for necessário, a repetição seria cansativa. Em vez disso, os tipos estrutura normalmente aparecem na declaração de um tipo nomeado, como **Employee**.

Um tipo nomeado **S** para estrutura não pode declarar um campo do mesmo tipo **S**: um valor agregado não pode conter a si mesmo. (Uma restrição análoga se aplica aos arrays.) Porém, **S** pode declarar um campo do tipo ponteiro ***S**, que nos permite criar estruturas de dados recursivas como listas ligadas e árvores. Isso é mostrado no código a seguir, que usa uma árvore binária para implementar uma ordenação por inserção (insertion sort):

gopl.io/ch4/treesort

```

type tree struct {
    value      int
    left, right *tree
}
// Sort ordena valores in-place.
func Sort(values []int) {
    var root *tree
    for _, v := range values {
        root = add(root, v)
    }
}

```

```

    appendValues(values[:0], root)
}

// appendValues concatena os elementos de t a values na ordem
// e devolve a fatia resultante
func appendValues(values []int, t *tree) []int {
    if t != nil {
        values = appendValues(values, t.left)
        values = append(values, t.value)
        values = appendValues(values, t.right)
    }
    return values
}

func add(t *tree, value int) *tree {
    if t == nil {
        // Equivalente a devolver &tree{value: value}
        t = new(tree)
        t.value = value
        return t
    }
    if value < t.value {
        t.left = add(t.left, value)
    } else {
        t.right = add(t.right, value)
    }
    return t
}

```

O valor zero de uma estrutura é composto dos valores zero para cada um de seus campos. Normalmente, é desejável que o valor zero seja um valor default natural ou sensato. Por exemplo, em **bytes.Buffer**, o valor inicial da estrutura é um buffer vazio pronto para uso, e o valor zero de **sync.Mutex**, que veremos no capítulo 9, é um mutex destravado, pronto para uso. Às vezes, esse comportamento inicial sensato acontece naturalmente, porém, em outras, o designer do tipo precisa trabalhar para consegui-lo.

O tipo estrutura sem campos é chamado de *estrutura vazia*, e é representado por **struct{}**. Tem tamanho zero e não carrega nenhuma informação, mas pode ser útil, apesar disso. Alguns programadores Go o usam no lugar de **bool** como o tipo do valor em um mapa que representa

um conjunto para enfatizar que somente as chaves são significativas, mas a economia de espaço é insignificante e a sintaxe é mais desajeitada, portanto, em geral, ele é evitado.

```
seen := make(map[string]struct{}) // conjunto de strings
// ...
if _, ok := seen[s]; !ok {
    seen[s] = struct{}{}
    // ...vendo s pela primeira vez...
}
```

4.4.1 Estruturas literais

Um valor de um tipo estrutura pode ser escrito por meio de uma *estrutura literal* que especifica valores para seus campos.

```
type Point struct{ X, Y int }
p := Point{1, 2}
```

Há duas formas de estrutura literal. A primeira forma, mostrada anteriormente, exige que um valor seja especificado para *cada* campo, na ordem correta. Isso sobrecarrega quem escreve (e quem lê), fazendo com que ele ou ela precise lembrar exatamente o que são os campos, e deixam o código frágil caso o conjunto de campos aumente ou seja reordenado posteriormente. Da mesma maneira, essa forma tende a ser usada somente no pacote que define o tipo da estrutura, ou em tipos menores de estrutura para os quais haja uma convenção óbvia para a ordem dos campos, como em `image.Point{x, y}` ou em `color.RGBA{red, green, blue, alpha}`.

A segunda forma é usada com mais frequência; nesse caso, um valor de estrutura é inicializado listando alguns ou todos os nomes dos campos e seus valores correspondentes, como na instrução a seguir do programa Lissajous da seção 1.4:

```
anim := gif.GIF{LoopCount: nframes}
```

Se um campo for omitido nesse tipo de literal, ele é definido com o valor zero de seu tipo. Como os nomes são fornecidos, a ordem dos campos não importa.

As duas formas não podem ser misturadas no mesmo literal. Também não podemos usar a primeira forma de literal (baseada na ordem) para passar por cima da regra segundo a qual identificadores não exportados não podem ser referenciados a partir de outro pacote.

```
package p
type T struct{ a, b int } // a e b não são exportados

package q
import "p"
var _ = p.T{a: 1, b: 2} // erro de compilação: não é possível
                        // referenciar a, b
var _ = p.T{1, 2}      // erro de compilação: não é possível
                        // referenciar a, b
```

Embora a última linha anterior não mencione os identificadores de campo não exportados, ela está realmente usando os respectivos campos, portanto a operação não é permitida.

Valores de estruturas podem ser passados como argumentos a funções e devolvidos a partir delas. Por exemplo, a função a seguir escala um **Point** de acordo com um fator especificado:

```
func Scale(p Point, factor int) Point {
    return Point{p.X * factor, p.Y * factor}
}

fmt.Println(Scale(Point{1, 2}, 5)) // "{5 10}"
```

Por questões de eficiência, tipos maiores de estrutura normalmente são passados ou devolvidos de funções indiretamente, usando um ponteiro,

```
func Bonus(e *Employee, percent int) int {
    return e.Salary * percent / 100
}
```

e isso é necessário se a função precisa modificar seu argumento, pois em uma linguagem que faz chamada por valor como Go, a função chamada recebe apenas uma cópia de um argumento, e não uma referência ao argumento original.

```
func AwardAnnualRaise(e *Employee) {
    e.Salary = e.Salary * 105 / 100
}
```

Como é comum lidar com estruturas por meio de ponteiros, é possível usar a notação abreviada a seguir para criar e inicializar uma variável **struct** e obter seu endereço:

```
pp := &Point{1, 2}
```

Essa linha é exatamente equivalente a:

```
pp := new(Point)
*pp = Point{1, 2}
```

mas **&Point{1, 2}** pode ser usado diretamente em uma expressão, por exemplo, em uma chamada de função.

4.4.2 Comparando estruturas

Se todos os campos de uma estrutura forem comparáveis, a estrutura em si será comparável, portanto duas expressões desse tipo podem ser comparadas usando **==** ou **!=**. A operação **==** compara os campos correspondentes de duas estruturas em ordem, desta forma as duas expressões de exibição a seguir são equivalentes:

```
type Point struct{ X, Y int }

p := Point{1, 2}
q := Point{2, 1}
fmt.Println(p.X == q.X && p.Y == q.Y) // "false"
fmt.Println(p == q)                  // "false"
```

Tipos de estrutura comparáveis, como qualquer outro tipo comparável, podem ser usados como o tipo das chaves de um mapa.

```
type address struct {
    hostname string
    port      int
}

hits := make(map[address]int)
hits[address{"golang.org", 443}]++
```

4.4.3 Inclusão de estruturas e campos anônimos

Nesta seção, veremos como o sistema incomum de *inclusão de estruturas* (struct embedding) de Go nos permite usar um tipo nomeado de

estrutura como um *campo anônimo* de outro tipo de estrutura, oferecendo um atalho sintático conveniente, de modo que uma expressão simples com ponto como `x.f` pode representar uma cadeia de campos como `x.d.e.f`.

Considere um programa de desenho 2-D que ofereça uma biblioteca de formas, como retângulos, elipses, estrelas e rodas. Estes são dois tipos que a biblioteca pode definir:

```
type Circle struct {
    X, Y, Radius int
}

type Wheel struct {
    X, Y, Radius, Spokes int
}
```

Um **Circle** (círculo) tem campos para as coordenadas **X** e **Y** de seu centro e um **Radius** (raio). Uma **Wheel** (roda) tem tudo que **Circle** tem, mais **Spokes**, que é o número de raios inscritos na roda. Vamos criar uma roda:

```
var w Wheel
w.X = 8
w.Y = 8
w.Radius = 5
w.Spokes = 20
```

À medida que o conjunto de formas aumenta, vemos semelhanças e repetições entre elas, portanto pode ser conveniente fatorar as partes comuns:

```
type Point struct {
    X, Y int
}

type Circle struct {
    Center Point
    Radius int
}

type Wheel struct {
    Circle Circle
    Spokes int
}
```

A aplicação pode ficar mais clara com isso, mas essa alteração deixa o acesso aos campos de **Wheel** mais extenso:

```
var w Wheel
w.Circle.Center.X = 8
w.Circle.Center.Y = 8
w.Circle.Radius = 5
w.Spokes = 20
```

Go nos permite declarar um campo com um tipo, mas sem nome; esses campos são chamados de *campos anônimos* (anonymous fields). O tipo do campo deve ser um tipo nomeado ou um ponteiro para um tipo nomeado. A seguir, **Circle** e **Wheel** têm um campo anônimo cada um. Dizemos que um **Point** está *incluído* (embedded) em **Circle**, e um **Circle** está incluído em **Wheel**.

```
type Circle struct {
    Point
    Radius int
}

type Wheel struct {
    Circle
    Spokes int
}
```

Graças à inclusão, podemos referenciar os nomes nos níveis mais fundos da hierarquia implícita sem fornecer os nomes intermediários.

```
var w Wheel
w.X = 8           // equivalente a w.Circle.Point.X = 8
w.Y = 8           // equivalente a w.Circle.Point.Y = 8
w.Radius = 5      // equivalente a w.Circle.Radius = 5
w.Spokes = 20
```

As formas explícitas exibidas nos comentários anteriores continuam válidas, mostrando que “campo anônimo” é uma espécie de termo impróprio. Os campos **Circle** e **Point** têm nomes – o mesmo do tipo nomeado – mas esses nomes são opcionais nas expressões com ponto. Podemos omitir alguns ou todos os campos anônimos ao selecionar seus subcampos.

Infelizmente, não há uma versão abreviada correspondente para a sintaxe

de estrutura literal, portanto nenhuma das linhas a seguir compila:

```
w = Wheel{8, 8, 5, 20} // erro de compilação: campos desconhecidos
w = Wheel{X: 8, Y: 8, Radius: 5, Spokes: 20} // erro de compilação: campos
// desconhecidos
```

A estrutura literal deve seguir o formato da declaração do tipo, com isso devemos usar uma das duas formas a seguir, que são equivalentes:

gopl.io/ch4/embed

```
w = Wheel{Circle{Point{8, 8}, 5}, 20}
w = Wheel{
    Circle: Circle{
        Point: Point{X: 8, Y: 8},
        Radius: 5,
    },
    Spokes: 20, // NOTA: a vírgula no final é necessária aqui (e em Radius)
}
fmt.Printf("%#v\n", w)
// Saída:
// Wheel{Circle:Circle{Point:Point{X:8, Y:8}, Radius:5}, Spokes:20}
w.X = 42
fmt.Printf("%#v\n", w)
// Saída:
// Wheel{Circle:Circle{Point:Point{X:42, Y:8}, Radius:5}, Spokes:20}
```

Observe como o advérbio **#** faz o verbo **%v** de **Printf** exibir valores em um formato semelhante à sintaxe de Go. Para valores de estrutura, essa forma inclui o nome de cada campo.

Como campos “anônimos” têm nomes implícitos, você não pode ter dois campos anônimos de mesmo tipo, pois seus nomes entrariam em conflito. Pelo fato de o nome do campo ser implicitamente determinado pelo seu tipo, o mesmo acontece com sua visibilidade. Nos exemplos anteriores, os campos anônimos **Point** e **Circle** são exportados. Se eles não fossem exportados (**point** e **circle**), ainda poderíamos usar o formato compacto:

```
w.X = 8 // equivalente a w.circle.point.X = 8
```

mas a forma longa explícita mostrada no comentário seria proibida fora

do pacote em que está a declaração, pois `circle` e `point` seriam inacessíveis.

O que vimos até agora sobre inclusão de estruturas é apenas uma pequena amostra da notação de ponto usada para selecionar campos de estrutura. Posteriormente, veremos que os campos anônimos não precisam ser estruturas; qualquer tipo nomeado ou ponteiro para um tipo nomeado pode ser usado. No entanto, por que você iria querer incluir um tipo que não tenha subcampos?

A resposta tem a ver com métodos. A notação abreviada utilizada para selecionar os campos de um tipo incluído funciona para selecionar seus métodos também. De fato, o tipo da estrutura mais externa ganha não só os campos do tipo incluído, mas também seus métodos. Esse sistema é a principal maneira pela qual comportamentos de objetos complexos são compostos de comportamentos mais simples. A *composição* é central na programação orientada a objetos em Go, e vamos explorá-la melhor na seção 6.3.

4.5 JSON

O JSON (JavaScript Object Notation, ou Notação de Objetos JavaScript) é uma notação padrão para enviar e receber informações estruturadas. O JSON não é a única notação desse tipo. O XML (seção 7.14), o ASN.1 e o Protocol Buffers do Google têm propósitos semelhantes e cada um tem seu nicho, mas por causa de sua simplicidade, legibilidade e do suporte universal, JSON é a notação mais amplamente utilizada.

Go tem um suporte excelente para codificação e decodificação desses formatos, oferecido pelos pacotes da biblioteca-padrão `encoding/json`, `encoding/xml`, `encoding/asn1`, e assim por diante, e todos esses pacotes têm APIs similares. Esta seção apresenta uma visão geral das partes mais importantes do pacote `encoding/json`.

JSON é uma codificação de valores JavaScript – strings, números, booleanos, arrays e objetos – como texto Unicode. É uma representação eficiente e legível para os tipos de dados básicos do capítulo 3 e os tipos

compostos deste capítulo – arrays, fatias, estruturas e mapas.

Os tipos básicos de JSON são números (em decimal ou notação científica), booleanos (**true** ou **false**) e strings, que são sequências de pontos de código Unicode entre aspas duplas, com escapes de barra invertida cuja notação é semelhante à de Go, embora os escapes numéricos `\uhhhh` de JSON representem códigos UTF-16, e não runas.

Esses tipos básicos podem ser combinados recursivamente usando arrays e objetos JSON. Um array JSON é uma sequência ordenada de valores, escrita como uma lista separada por vírgulas, entre colchetes; arrays JSON são usados para codificar arrays e fatias em Go. Um objeto JSON é um mapeamento de strings para valores, escrito como uma sequência de pares **name:value** separados por vírgulas e entre chaves; objetos JSON são usados para codificar mapas (com chaves do tipo string) e estruturas em Go. Por exemplo:

boolean	true
number	-273.15
string	"She said \"Hello, 世界\""
array	["gold", "silver", "bronze"]
object	{"year": 1980, "event": "archery", "medals": ["gold", "silver", "bronze"]}

Considere uma aplicação que reúna avaliações de filmes e ofereça recomendações. Seu tipo de dado **Movie** e uma lista típica de valores estão declarados a seguir. [As strings literais após as declarações dos campos **Year** e **Color** são *etiquetas de campos* (field tags), que serão explicadas em breve.]

gopl.io/ch4/movie

```
type Movie struct {
    Title string
    Year  int  `json:"released"`
    Color bool `json:"color,omitempty"`
    Actors []string
}

var movies = []Movie{
    {Title: "Casablanca", Year: 1942, Color: false,
```

```

    Actors: []string{"Humphrey Bogart", "Ingrid Bergman"}},
    {Title: "Cool Hand Luke", Year: 1967, Color: true,
      Actors: []string{"Paul Newman"}},
    {Title: "Bullitt", Year: 1968, Color: true,
      Actors: []string{"Steve McQueen", "Jacqueline Bisset"}},
    // ...
}

```

Estruturas de dados como essa são excelentes para JSON, e é fácil fazer a conversão nos dois sentidos. Converter uma estrutura de dados Go como **movies** para JSON chama-se *marshaling* (semelhante a serialização). O marshaling é feito por **json.Marshal**:

```

data, err := json.Marshal(movies)
if err != nil {
    log.Fatalf("JSON marshaling failed: %s", err)
}
fmt.Printf("%s\n", data)

```

Marshal produz uma fatia de bytes contendo uma string bem longa sem espaços em branco extras; usamos mais de uma linha para que os dados pudessem ser mostrados:

```

[{"Title":"Casablanca","released":1942,"Actors":["Humphrey Bogart","Ingrid Bergman"]}, {"Title":"Cool Hand Luke","released":1967,"color":true,"Actors":["Paul Newman"]}, {"Title":"Bullitt","released":1968,"color":true,"Actors":["Steve McQueen","Jacqueline Bisset"]}

```

Essa representação compacta contém todas as informações, mas é difícil de ler. Para consumo humano, uma variante chamada **json.MarshalIndent** produz uma saída elegantemente indentada. Dois argumentos adicionais definem um prefixo para cada linha da saída e uma string para cada nível de indentação:

```

data, err := json.MarshalIndent(movies, "", "  ")
if err != nil {
    log.Fatalf("JSON marshaling failed: %s", err)
}
fmt.Printf("%s\n", data)

```

O código anterior exibe o seguinte:

```

[
  {
    "Title": "Casablanca",

```

```

    "released": 1942,
    "Actors": [
        "Humphrey Bogart",
        "Ingrid Bergman"
    ]
},
{
    "Title": "Cool Hand Luke",
    "released": 1967,
    "color": true,
    "Actors": [
        "Paul Newman"
    ]
},
{
    "Title": "Bullitt",
    "released": 1968,
    "color": true,
    "Actors": [
        "Steve McQueen",
        "Jacqueline Bisset"
    ]
}
]

```

O marshaling usa os nomes dos campos de uma estrutura Go como os nomes dos campos de objetos JSON (por meio de *reflexão*, como veremos na seção 12.6). Pode haver marshaling somente de campos exportados, motivo pelo qual escolhemos nomes com letras maiúsculas para todos os nomes de campo em Go.

Talvez você tenha percebido que o nome do campo **Year** mudou para **released** na saída, e **Color** mudou para **color**. Isso se deve às *etiquetas de campos* (field tags). Uma etiqueta de campo é uma estrutura de metadados associada em tempo de compilação ao campo de uma estrutura:

```

Year int `json:"released"`
Color bool `json:"color,omitempty"`

```

Uma etiqueta de campo pode ser qualquer string literal, mas é convencionalmente interpretada como uma lista de pares **key:"value"**

separados por espaço; como contêm aspas duplas, as etiquetas de campo normalmente são escritas como strings literais brutas (*raw string literals*). A chave `json` controla o comportamento do pacote `encoding/json`, e outros pacotes `encoding/...` seguem essa convenção. A primeira parte da etiqueta de campo `json` especifica um nome JSON alternativo para o campo em Go. Etiquetas de campo frequentemente são usadas para especificar um nome JSON idiomático como `total_count` para um campo em Go chamado `TotalCount`. A etiqueta para `Color` tem uma opção adicional, `omitempty`, que indica que nenhuma saída JSON deve ser gerada se o campo tiver o valor zero de seu tipo (`false`, nesse caso) ou se estiver vazio. De fato, a saída JSON de *Casablanca*, um filme em preto e branco, não tem um campo `color`.

A operação inversa de marshaling, que é decodificar JSON e preencher uma estrutura de dados Go, chama-se *unmarshaling*, e é feita por `json.Unmarshal`. O código a seguir faz o unmarshaling dos dados JSON de filmes para uma fatia de estruturas cujo único campo é `Title`. Ao definir estruturas de dados adequadas em Go dessa maneira, podemos selecionar quais partes da entrada JSON serão decodificadas e quais serão descartadas. Quando `Unmarshal` retornar, ela terá preenchido a fatia com a informação `Title`; outros nomes nos dados JSON serão ignorados.

```
var titles []struct{ Title string }
if err := json.Unmarshal(data, &titles); err != nil {
    log.Fatalf("JSON unmarshaling failed: %s", err)
}
fmt.Println(titles) // "[{Casablanca} {Cool Hand Luke} {Bullitt}]"
```

Muitos web services oferecem uma interface JSON – faça uma requisição com HTTP e as informações desejadas serão devolvidas em formato JSON. Para ilustrar, vamos consultar o sistema de acompanhamento de problemas do GitHub (GitHub issue tracker) usando sua interface de web service. Inicialmente, definiremos os tipos e as constantes necessários:

gopl.io/ch4/github

```
// O pacote github disponibiliza uma API Go para o sistema de
// acompanhamento de problemas do GitHub
// Veja https://developer.github.com/v3/search/#search-issues
```

```

package github

import "time"

const IssuesURL = "https://api.github.com/search/issues"

type IssuesSearchResult struct {
    TotalCount int `json:"total_count"`
    Items      []*Issue
}

type Issue struct {
    Number      int
    HTMLURL     string `json:"html_url"`
    Title       string
    State       string
    User        *User
    CreatedAt   time.Time `json:"created_at"`
    Body        string    // em formato Markdown
}

type User struct {
    Login      string
    HTMLURL    string `json:"html_url"`
}

```

Como já vimos, os nomes de todos os campos das estruturas devem começar com letra maiúscula, mesmo que seus nomes JSON não as usem. No entanto, a lógica que associa nomes JSON a nomes de estruturas Go durante o unmarshaling não diferencia letras maiúsculas de minúsculas, portanto só é necessário usar uma etiqueta de campo quando houver um underscore no nome JSON, mas não no nome Go. Novamente, estamos sendo seletivos sobre quais campos serão decodificados; a resposta da pesquisa no GitHub contém muito mais informações do que mostramos aqui.

A função **SearchIssues** faz uma requisição HTTP e decodifica o resultado como JSON. Como os termos da consulta apresentados por um usuário podem conter caracteres como **?** e **&**, que têm significados especiais em um URL, usamos **url.QueryEscape** para garantir que eles sejam interpretados literalmente.

gopl.io/ch4/github

```

package github

```

```

import (
    "encoding/json"
    "fmt"
    "net/http"
    "net/url"
    "strings"
)

// SearchIssues faz uma consulta ao sistema de acompanhamento de problemas
// do GitHub
func SearchIssues(terms []string) (*IssuesSearchResult, error) {
    q := url.QueryEscape(strings.Join(terms, " "))
    resp, err := http.Get(IssuesURL + "?q=" + q)
    if err != nil {
        return nil, err
    }

    // Devemos fechar resp.Body em todos os paths de execução
    // (O capítulo 5 apresenta 'defer', que simplificará isto)
    if resp.StatusCode != http.StatusOK {
        resp.Body.Close()
        return nil, fmt.Errorf("search query failed: %s", resp.Status)
    }

    var result IssuesSearchResult
    if err := json.NewDecoder(resp.Body).Decode(&result); err != nil {
        resp.Body.Close()
        return nil, err
    }
    resp.Body.Close()
    return &result, nil
}

```

Os exemplos anteriores usaram `json.Unmarshal` para decodificar todo o conteúdo de uma fatia de bytes como uma única entidade JSON. Para variar, esse exemplo usa o decodificador de *streaming*, `json.Decoder`, que permite que várias entidades JSON sejam decodificadas em sequência a partir do mesmo stream, apesar de não precisarmos desse recurso aqui. Como esperado, há um codificador de streaming correspondente chamado `json.Encoder`.

A chamada a `Decode` preenche a variável `result`. Há várias maneiras de formatar seu valor de forma elegante. A maneira mais simples, mostrada

por meio do comando **issues** a seguir, é na forma de uma tabela de texto com colunas de largura fixa, mas na próxima seção veremos uma abordagem mais sofisticada baseada em templates.

gopl.io/ch4/issues

```
// Issues exibe tabela de problemas do GitHub que correspondem aos termos
// de pesquisa
package main

import (
    "fmt"
    "log"
    "os"
    "gopl.io/ch4/github"
)

func main() {
    result, err := github.SearchIssues(os.Args[1:])
    if err != nil {
        log.Fatal(err)
    }
    fmt.Printf("%d issues:\n", result.TotalCount)
    for _, item := range result.Items {
        fmt.Printf("#%-5d %9.9s %.55s\n",
            item.Number, item.User.Login, item.Title)
    }
}
```

Os argumentos de linha de comando especificam os termos da pesquisa. O comando a seguir consulta o sistema de acompanhamento de problemas do projeto Go em busca da lista de bugs abertos relacionados à decodificação JSON:

```
$ go build gopl.io/ch4/issues
$ ./issues repo:golang/go is:open json decoder
13 issues:
#5680 eaigner encoding/json: set key converter on en/decoder
#6050 gopherbot encoding/json: provide tokenizer
#8658 gopherbot encoding/json: use bufio
#8462 kortschak encoding/json: UnmarshalText confuses json.Unmarshal
#5901 rsc encoding/json: allow override type marshaling
#9812 klauspost encoding/json: string tag not symmetric
#7872 extempora encoding/json: Encoder internally buffers full output
```

```
#9650 cespere encoding/json: Decoding gives errPhase when unmarshalin
#6716 gopherbot encoding/json: include field name in unmarshal error me
#6901 lukescott encoding/json, encoding/xml: option to treat unknown fi
#6384 joeshaw encoding/json: encode precise floating point integers u
#6647 btracey x/tools/cmd/godoc: display type kind of each named type
#4237 gjemiller encoding/base64: URLEncoding padding is optional
```

A interface de web service do GitHub em <https://developer.github.com/v3/> tem tantos recursos que não há espaço para mostrar todos aqui.

Exercício 4.10: Modifique `issues` para informar os resultados em termos de idade, por exemplo, menos de um mês, menos de um ano e mais de um ano.

Exercício 4.11: Crie uma ferramenta que permita aos usuários criar, ler, atualizar e fechar *issues* do GitHub a partir da linha de comando, chamando seu editor de texto preferido quando houver necessidade de fornecer uma quantidade substancial de texto de entrada.

Exercício 4.12: O popular web comic (quadrinhos da web) *xkcd* tem uma interface JSON. Por exemplo, uma requisição para <https://xkcd.com/571/info.0.json> gera uma descrição detalhada do quadrinho 571, um de meus vários favoritos. Faça download de cada URL (uma vez!) e crie um índice offline. Crie uma ferramenta *xkcd* que, usando esse índice, exiba o URL e transcreva cada quadrinho que corresponda a um termo de pesquisa fornecido na linha de comando.

Exercício 4.13: O web service baseado em JSON do Open Movie Database permite pesquisar em <https://omdbapi.com/> em busca de um filme de acordo com o nome e fazer download da imagem do pôster. Crie uma ferramenta *poster* que faça download da imagem do pôster para o filme especificado na linha de comando.

4.6 Templates de texto e HTML

O exemplo anterior faz apenas a mais simples formatação possível, para a qual `Printf` é totalmente adequado. Porém, às vezes, a formatação precisa

ser mais sofisticada, e é desejável separar totalmente o formato do código. Isso pode ser feito com os pacotes `text/template` e `html/template`, que oferecem um sistema para substituir os valores de variáveis em um template de texto ou HTML.

Um template é uma string ou um arquivo contendo uma ou mais partes entre chaves duplas, `{{...}}`, chamadas *ações*. A maior parte da string é exibida literalmente, mas as ações disparam outros comportamentos. Cada ação contém uma expressão na linguagem do template: uma notação simples, porém eficaz, para exibir valores, selecionar campos de estruturas, chamar funções e métodos, expressar controle de fluxo como instruções `if-else` e loops `range` e instanciar outros templates. Uma string com um template simples é apresentada a seguir:

gopl.io/ch4/issuesreport

```
const templ = `{{.TotalCount}} issues:
{{range .Items}}-----
Number: {{.Number}}
User:   {{.User.Login}}
Title:  {{.Title | printf "%.64s"}}
Age:    {{.CreatedAt | daysAgo}} days
{{end}}`
```

Esse template inicialmente exibe a quantidade de problemas correspondentes e então exibe o número, o usuário, o título e a idade em dias de cada problema. Em uma ação, há uma noção de valor atual, referenciado como “ponto” e escrito como “.”. O ponto inicialmente refere-se ao parâmetro do template, que será um `github.IssuesSearchResult` nesse exemplo. A ação `{{.TotalCount}}` é expandida com o valor do campo `TotalCount`, exibido da maneira usual. As ações `{{range .Items}}` e `{{end}}` criam um loop, portanto o texto entre eles é expandido várias vezes, com o ponto vinculado a elementos sucessivos de `Items`.

Em uma ação, a notação `|` transforma o resultado de uma operação no argumento de outra, de modo análogo a um pipeline no shell Unix. No caso de `Title`, a segunda operação é a função `printf`, uma função

embutida que é sinônimo de `fmt.Sprintf` em todos os templates. Para **Age**, a segunda operação é a função seguinte, **daysAgo**, que converte o campo **CreatedAt** em um tempo decorrido usando **time.Since**:

```
func daysAgo(t time.Time) int {  
    return int(time.Since(t).Hours() / 24)  
}
```

Observe que o tipo de **CreatedAt** é **time.Time**, e não **string**. Do mesmo modo que um tipo pode controlar sua formatação em string (seção 2.5) ao definir determinados métodos, um tipo também pode definir métodos para controlar seu comportamento de marshaling e de unmarshaling JSON. O valor JSON após um marshaling de um **time.Time** é uma string em um formato padrão.

Gerar uma saída com um template é um processo de dois passos. Inicialmente, devemos fazer parse do template em uma representação interna adequada e então executá-la com entradas específicas. O parsing precisa ser feito apenas uma vez. O código a seguir cria e faz parse do template **templ** definido anteriormente. Observe o encadeamento das chamadas de métodos: **template.New** cria e devolve um template; **Funcs** adiciona **daysAgo** ao conjunto de funções acessíveis nesse template e então o retorna; por fim, **Parse** é chamado no resultado.

```
report, err := template.New("report").  
    Funcs(template.FuncMap{"daysAgo": daysAgo}).  
    Parse(templ)  
if err != nil {  
    log.Fatal(err)  
}
```

Como os templates normalmente são fixos em tempo de compilação, uma falha no parse do template indica um bug fatal no programa. A função auxiliar **template.Must** deixa o tratamento de erros mais conveniente: ela aceita um template e um erro, verifica se o erro é nil (e gera pânico, se não for) e então devolve o template. Retomaremos essa ideia na seção 5.9.

Depois que o template for criado, expandido com **daysAgo**, interpretado e conferido, podemos executá-lo usando um **github.IssuesSearchResult** como fonte de dados e **os.Stdout** como destino:

```

var report = template.Must(template.New("issuelist").
    Funcs(template.FuncMap{"daysAgo": daysAgo}).
    Parse(templ))

func main() {
    result, err := github.SearchIssues(os.Args[1:])
    if err != nil {
        log.Fatal(err)
    }
    if err := report.Execute(os.Stdout, result); err != nil {
        log.Fatal(err)
    }
}

```

O programa exibe um relatório em formato texto simples como este:

```

$ go build gopl.io/ch4/issuesreport
$ ./issuesreport repo:golang/go is:open json decoder
13 issues:
-----
Number: 5680
User:   eaigner
Title:  encoding/json: set key converter on en/decoder
Age:    750 days
-----
Number: 6050
User:   gopherbot
Title:  encoding/json: provide tokenizer
Age:    695 days
-----
...

```

Vamos agora passar para o pacote **html/template**. Ele usa a mesma API e a linguagem de expressão de **text/template**, mas acrescenta recursos para escaping automático de strings, que aparecem em HTML, JavaScript, CSS ou URLs, conforme a sintaxe apropriada ao contexto. Esses recursos podem ajudar a evitar um problema constante de segurança na geração de HTML – um *ataque de injeção* – em que um inimigo compõe um valor de string como o título de um problema para incluir um código malicioso que, quando escapado indevidamente por um template, dá a ele o controle sobre a página.

O template a seguir exibe a lista de problemas na forma de uma tabela

HTML. Observe a importação diferente:

gopl.io/ch4/issueshtml

```
import "html/template"

var issueList = template.Must(template.New("issuelist").Parse(`
<h1>{{.TotalCount}} issues</h1>
<table>
<tr style='text-align: left'>
    <th>#</th>
    <th>State</th>
    <th>User</th>
    <th>Title</th>
</tr>
{{range .Items}}
<tr>
    <td><a href='{{.HTMLURL}}'>{{.Number}}</a></td>
    <td>{{.State}}</td>
    <td><a href='{{.User.HTMLURL}}'>{{.User.Login}}</a></td>
    <td><a href='{{.HTMLURL}}'>{{.Title}}</a></td>
</tr>
{{end}}
</table>
`))
```

O comando a seguir executa o novo template nos resultados de uma consulta um pouco diferente:

```
$ go build gopl.io/ch4/issueshtml
$ ./issueshtml repo:golang/go commenter:gopherbot json encoder >issues.html
```

A figura 4.4 mostra a aparência da tabela em um navegador web. Os links se conectam às páginas web apropriadas no GitHub.

#	State	User	Title
7872	open	extemporalgenome	encoding/json: Encoder internally buffers full output
5683	open	gopherbot	encoding/json: performance slower than expected
6901	open	lukescott	encoding/json, encoding/xml: option to treat unknown fields as an error
4474	closed	gopherbot	encoding/json: json encoder fails for embedded non-struct fields
4747	closed	gopherbot	encoding/json Added tag options to ignore fields of struct for encoder/decoder separately
7767	closed	gopherbot	encoding/json: Encoder adds trailing newlines
4606	closed	gopherbot	JSON Package fails to properly escape strings
8582	closed	matt-duch	encoding/json: inconsistent behavior in *(numeric type) and string tag option
6339	closed	gopherbot	encoding/json: Marshal of nil net.IP fails
7337	closed	gopherbot	encoding/json: make "json" tag user-settable
11508	closed	josharian	cmd/go: trace http viewer: "http: multiple response.WriteHeader calls"
1017	closed	gopherbot	json crash on {} input
8592	closed	gopherbot	encoding/json: No way to avoid HTMLEscape when Marshal()-ing
7846	closed	gopherbot	encoding/json: Slice created using reflect.MakeSlice() treated as interface{}
2761	closed	gopherbot	Marshaler cannot work with omitempty in encoding/json
1133	closed	gopherbot	encoding/asn1: inconsistent APIs
7841	closed	gopherbot	reflect: reflect.unpackEface reflect/value.go:174 unexpected fault address 0x0

Figura 4.4 – Tabela HTML dos problemas do projeto Go relacionados à codificação JSON.

Nenhum dos problemas da figura 4.4 representa um desafio para HTML, mas podemos ver o efeito mais claramente com os problemas cujos títulos contêm metacaracteres HTML como `&` e `<`. Selecionamos dois desses problemas neste exemplo:

```
$ ./issueshtml repo:golang/go 3133 10535 >issues2.html
```

A figura 4.5 mostra o resultado dessa consulta. Observe que o pacote `html/template` inseriu escapes HTML automaticamente nos títulos para que eles aparecessem literalmente. Se tivéssemos usado o pacote `text/template` por engano, a string de quatro caracteres `"<"` teria sido renderizada como um caractere menor que, ou seja, `'<'`, e a string `"<link>"` teria se transformado em um elemento `link`, alterando a estrutura do documento HTML e, quem sabe, comprometendo sua segurança.

Podemos eliminar esse comportamento de autoescaping para campos que contenham dados HTML confiáveis usando o tipo nomeado de string `template.HTML` no lugar de `string`. Tipos nomeados semelhantes existem para JavaScript, CSS e URLs confiáveis. O programa a seguir demonstra o princípio usando dois campos de mesmo valor, porém com tipos diferentes: **A** é uma string e **B** é um `template.HTML`.



Figura 4.5 – Metacaracteres HTML em títulos de problemas são exibidos corretamente.

gopl.io/ch4/autoescape

```
func main() {
    const templ = `

A: {{.A}}



B: {{.B}}

`
    t := template.Must(template.New("escape").Parse(templ))
    var data struct {
        A string          // texto simples não confiável
        B template.HTML    // HTML confiável
    }
    data.A = "<b>Hello!</b>"
    data.B = "<b>Hello!</b>"
    if err := t.Execute(os.Stdout, data); err != nil {
        log.Fatal(err)
    }
}
```

A figura 4.6 mostra a saída do template conforme apresentada em um navegador. Podemos ver que **A** foi sujeito a escaping, mas **B** não foi.

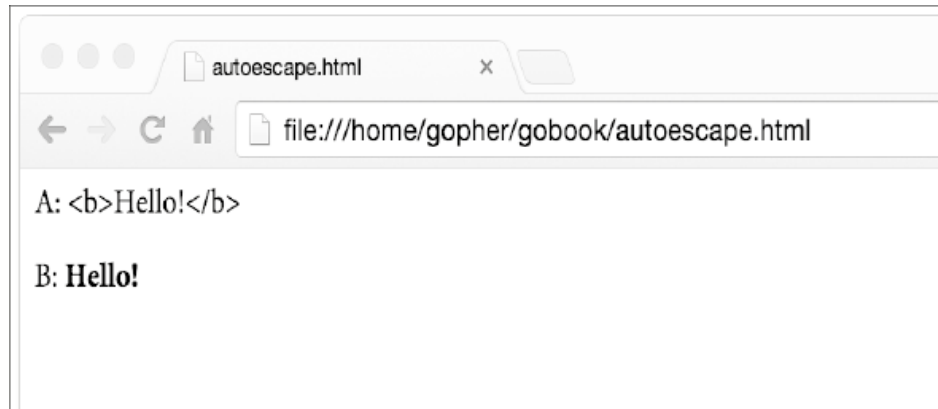


Figura 4.6 – Valores de string são escapados para HTML, mas valores `template.HTML` não são.

Temos espaço aqui para mostrar somente os recursos mais básicos do sistema de template. Como sempre, para obter mais informações, consulte a documentação do pacote:

```
$ go doc text/template
$ go doc html/template
```

Exercício 4.14: Crie um servidor web que faça consultas no GitHub uma vez e então permita navegar pela lista de bugs informados, milestones (marcos) e usuários.

5

Funções

Uma função permite reunir uma sequência de instruções como uma unidade que pode ser chamada de outro ponto de um programa, talvez várias vezes. Funções possibilitam dividir uma tarefa grande em partes menores que podem muito bem ser escritas por pessoas diferentes, separadas no tempo e no espaço. Uma função oculta seus detalhes de implementação dos usuários. Por todos esses motivos, as funções são uma parte crucial de qualquer linguagem de programação.

Já vimos várias funções. Vamos agora investir nosso tempo em uma discussão mais completa. O exemplo exposto neste capítulo é de um web crawler, isto é, o componente de uma ferramenta de pesquisa web responsável por buscar páginas web, descobrir seus links, buscar as páginas identificadas por esses links, e assim por diante. Um web crawler oferece amplas oportunidades para explorar recursão, funções anônimas, tratamento de erros e aspectos de funções que são exclusivos de Go.

5.1 Declarações de funções

Uma declaração de função tem um nome, uma lista de parâmetros, uma lista opcional de resultados e um corpo:

```
func nome(lista-de-parâmetros) (lista-de-resultados) {  
    corpo  
}
```

A lista de parâmetros especifica os nomes e os tipos dos *parâmetros* da função, que são as variáveis locais cujos valores ou *argumentos* são fornecidos por quem chama a função. A lista de resultados especifica os tipos dos valores devolvidos pela função. Se a função devolver um resultado sem nome ou não devolver resultado algum, os parênteses são

opcionais e, geralmente, omitidos. Omitir totalmente a lista de resultados declara uma função que não retorna nenhum valor e é chamada apenas pelos seus efeitos. Na função **hypot**,:

```
func hypot(x, y float64) float64 {  
    return math.Sqrt(x*x + y*y)  
}  
fmt.Println(hypot(3, 4)) // "5"
```

x e **y** são parâmetros na declaração, **3** e **4** são argumentos da chamada e a função devolve um valor do tipo **float64**.

Assim como os parâmetros, os resultados podem ser nomeados. Nesse caso, cada nome declara uma variável local inicializada com o valor zero de seu tipo.

Uma função que tenha uma lista de resultados deve terminar com uma instrução **return**, a menos que a execução claramente não possa alcançar o final da função, talvez porque termine com uma chamada a **panic** ou com um loop infinito **for** sem **break**.

Como vimos em **hypot**, uma sequência de parâmetros ou resultados de mesmo tipo pode ser fatorada para que o tipo propriamente dito seja escrito apenas uma vez. As duas declarações a seguir são equivalentes:

```
func f(i, j, k int, s, t string)          { /* ... */ }  
func f(i int, j int, k int, s string, t string) { /* ... */ }
```

A seguir, estão quatro maneiras de declarar uma função com dois parâmetros e um resultado, todos do tipo **int**. O identificador vazio pode ser usado para enfatizar que um parâmetro não é usado.

```
func add(x int, y int) int { return x + y }  
func sub(x, y int) (z int) { z = x - y; return }  
func first(x int, _ int) int { return x }  
func zero(int, int) int { return 0 }  
  
fmt.Printf("%T\n", add)    // "func(int, int) int"  
fmt.Printf("%T\n", sub)    // "func(int, int) int"  
fmt.Printf("%T\n", first)  // "func(int, int) int"  
fmt.Printf("%T\n", zero)   // "func(int, int) int"
```

O tipo de uma função às vezes é chamado de sua *assinatura*. Duas funções têm o mesmo tipo ou assinatura se tiverem a mesma sequência de

tipos de parâmetro e a mesma sequência de tipos de resultado. Os nomes dos parâmetros e dos resultados não afetam o tipo, nem o fato de terem sido declarados usando a forma fatorada.

Toda chamada de função deve especificar um argumento para cada parâmetro, na ordem em que os parâmetros foram declarados. Go não tem conceito de valores default de parâmetros nem maneira de especificar argumentos pelo nome, portanto os nomes dos parâmetros e os resultados não importam a quem chama a função, exceto como documentação.

Os parâmetros são variáveis locais no corpo da função, com seus valores iniciais definidos com os argumentos fornecidos por quem chamou a função. Parâmetros de função e resultados nomeados são variáveis no mesmo bloco léxico das variáveis locais mais externas da função.

Argumentos são passados *por valor*, portanto a função recebe uma cópia de cada argumento; modificações na cópia não afetam quem chama a função. No entanto, se o argumento contiver algum tipo de referência, por exemplo, um ponteiro, uma fatia, um mapa, uma função ou um canal, quem chama a função poderá ser afetado por qualquer modificação que a função fizer às variáveis referenciadas *indiretamente* pelo argumento.

Ocasionalmente, você poderá deparar com uma declaração de função sem corpo, indicando que a função está implementada em uma linguagem diferente de Go. Uma declaração desse tipo define a assinatura da função.

```
package math  
  
func Sin(x float64) float64 // implementada em linguagem assembly
```

5.2 Recursão

Funções podem ser *recursivas*, isto é, podem chamar a si mesmas direta ou indiretamente. A recursão é uma técnica eficaz para muitos problemas e é essencial para processar estruturas de dados recursivas. Na seção 4.4, usamos recursão em uma árvore para implementar uma ordenação simples por inserção. Nesta seção, usaremos a recursão novamente para

processar documentos HTML.

O programa de exemplo a seguir usa um pacote não padrão, golang.org/x/net/html, que oferece um parser HTML. Os repositórios golang.org/x/... armazenam pacotes projetados e mantidos pela equipe de Go para aplicações como rede, processamento de textos internacionalizados, plataformas móveis, manipulação de imagens, criptografia e ferramentas de desenvolvedores. Esses pacotes não constam na biblioteca-padrão porque ainda estão em desenvolvimento ou porque raramente são necessários à maioria dos programadores Go.

As partes da API golang.org/x/net/html de que precisaremos estão a seguir. A função `html.Parse` lê uma sequência de bytes, faz seu parse e devolve a raiz da árvore do documento HTML, que é um `html.Node`. O HTML tem vários tipos de nós – textos, comentários, e assim por diante – mas, neste caso, estamos interessados somente nos nós de *elementos* na forma `<name key='value'>`.

golang.org/x/net/html

```
package html

type Node struct {
    Type           NodeType
    Data           string
    Attr           []Attribute
    FirstChild, NextSibling *Node
}

type NodeType int32

const (
    ErrorNode NodeType = iota
    TextNode
    DocumentNode
    ElementNode
    CommentNode
    DoctypeNode
)

type Attribute struct {
    Key, Val string
}

func Parse(r io.Reader) (*Node, error)
```

A função **main** faz parse da entrada-padrão como HTML, extrai os links usando uma função recursiva **visit** e exibe cada link identificado:

gopl.io/ch5/findlinks1

```
// Findlinks1 exibe os links de um documento HTML lido da entrada-padrão.
package main

import (
    "fmt"
    "os"

    "golang.org/x/net/html"
)

func main() {
    doc, err := html.Parse(os.Stdin)
    if err != nil {
        fmt.Fprintf(os.Stderr, "findlinks1: %v\n", err)
        os.Exit(1)
    }
    for _, link := range visit(nil, doc) {
        fmt.Println(link)
    }
}
```

A função **visit** percorre uma árvore de nós HTML, extrai o link do atributo **href** de cada elemento *âncora* `<a href='... '>`, concatena os links em uma fatia de strings e devolve a fatia resultante:

```
// visit concatena cada link encontrado em n a links e devolve o resultado.
func visit(links []string, n *html.Node) []string {
    if n.Type == html.ElementNode && n.Data == "a" {
        for _, a := range n.Attr {
            if a.Key == "href" {
                links = append(links, a.Val)
            }
        }
    }
    for c := n.FirstChild; c != nil; c = c.NextSibling {
        links = visit(links, c)
    }
    return links
}
```

Para descer na árvore para um nó **n**, **visit** chama a si mesma

recursivamente para cada um dos filhos de `n`, armazenados na lista ligada `FirstChild`.

Vamos executar `findlinks` na página inicial de Go, fazendo pipe da saída de `fetch` (seção 1.5) para a entrada de `findlinks`. Editamos um pouco a saída por questões de concisão.

```
$ go build gopl.io/ch1/fetch
$ go build gopl.io/ch5/findlinks1
$ ./fetch https://golang.org | ./findlinks1
#
/doc/
/pkg/
/help/
/blog/
http://play.golang.org/
//tour.golang.org/
https://golang.org/dl/
//blog.golang.org/
/LICENSE
/doc/tos.html
http://www.google.com/intl/en/policies/privacy/
```

Observe a variedade de formas de links que estão na página. Mais adiante, veremos como resolvê-los em relação ao URL base, `https://golang.org`, para criar URLs absolutos.

O próximo programa usa recursão na árvore de nós HTML para exibir a estrutura da árvore. À medida que cada elemento é encontrado, o programa insere a tag do elemento em uma pilha e, em seguida, exibe a pilha.

gopl.io/ch5/outline

```
func main() {
    doc, err := html.Parse(os.Stdin)
    if err != nil {
        fmt.Fprintf(os.Stderr, "outline: %v\n", err)
        os.Exit(1)
    }
    outline(nil, doc)
}

func outline(stack []string, n *html.Node) {
```

```

    if n.Type == html.ElementNode {
        stack = append(stack, n.Data) // faz push da tag
        fmt.Println(stack)
    }
    for c := n.FirstChild; c != nil; c = c.NextSibling {
        outline(stack, c)
    }
}

```

Observe uma sutileza: embora **outline** faça “push” (inserção) de um elemento em **stack**, não há um pop (remoção) correspondente. Quando **outline** chama a si mesma recursivamente, quem foi chamado recebe uma cópia de **stack**. Embora a função chamada possa concatenar elementos a essa fatia, modificar seu array subjacente e, quem sabe, até mesmo alocar um novo array, ela não modifica os elementos iniciais visíveis a quem chamou, portanto, quando a função retorna, a **stack** de quem chamou estará como antes da chamada.

Eis o esquema de <https://golang.org>, novamente editado por questões de concisão:

```

$ go build gopl.io/ch5/outline
$ ./fetch https://golang.org | ./outline
[html]
[html head]
[html head meta]
[html head title]
[html head link]
[html body]
[html body div]
[html body div]
[html body div div]
[html body div div form]
[html body div div form div]
[html body div div form div a]
...

```

Como você pode notar por meio dos experimentos com **outline**, a maioria dos documentos HTML pode ser processada com apenas alguns níveis de recursão, mas não é difícil criar páginas web patológicas, que exijam uma recursão extremamente profunda.

Muitas implementações de linguagens de programação usam uma pilha de chamada de funções de tamanho fixo; tamanhos de 64 KB a 2 MB são típicos. Pilhas de tamanho fixo impõem um limite na profundidade da recursão, portanto devemos ter cuidado para evitar um *stack overflow* (estouro de pilha) ao percorrer estruturas grandes de dados recursivamente. Pilhas de tamanho fixo podem até mesmo representar um risco à segurança. Em comparação, implementações típicas de Go usam pilhas de tamanho variável que começam pequenas e crescem à medida que for necessário, até um limite da ordem de um gigabyte. Isso permite usar recursão de forma segura, sem nos preocuparmos com estouro de pilha.

Exercício 5.1: Altere o programa `findlinks` para que percorra a lista ligada `n.FirstChild` usando chamadas recursivas a `visit` no lugar de um loop.

Exercício 5.2: Escreva uma função para preencher um mapeamento de nomes de elementos – `p`, `div`, `span`, e assim por diante – para o número de elementos com esse nome em uma árvore de documento HTML.

Exercício 5.3: Escreva uma função para exibir o conteúdo de todos os nós de texto em uma árvore de documento HTML. Não desça em elementos `<script>` ou `<style>`, pois seus conteúdos não são visíveis em um navegador web.

Exercício 5.4: Estenda a função `visit` para que ela extraia outros tipos de links do documento, como imagens, scripts e folhas de estilo.

5.3 Múltiplos valores de retorno

Uma função pode devolver mais de um resultado. Vimos muitos exemplos de funções de pacotes padrões que devolvem dois valores: o resultado desejado do processamento e um valor de erro ou um booleano que informa se o processamento funcionou. O exemplo a seguir mostra como escrever uma dessas funções.

O programa a seguir é uma variação de `findlinks` que faz sua própria requisição HTTP; assim, não precisamos mais executar `fetch`. Como as

operações de HTTP e de parsing podem falhar, **findLinks** declara dois resultados: a lista de links identificada e um erro. A propósito, o parser HTML normalmente é capaz de se recuperar de uma entrada ruim e criar um documento contendo nós de erro, portanto **Parse** raramente falha; quando isso ocorre, em geral, é por causa de erros subjacentes de E/S.

gopl.io/ch5/findlinks2

```
func main() {
    for _, url := range os.Args[1:] {
        links, err := findLinks(url)
        if err != nil {
            fmt.Fprintf(os.Stderr, "findlinks2: %v\n", err)
            continue
        }
        for _, link := range links {
            fmt.Println(link)
        }
    }
}

// findLinks faz uma requisição HTTP GET para url, faz parse da
// resposta como HTML, extrai e devolve os links.
func findLinks(url string) ([]string, error) {
    resp, err := http.Get(url)
    if err != nil {
        return nil, err
    }
    if resp.StatusCode != http.StatusOK {
        resp.Body.Close()
        return nil, fmt.Errorf("getting %s: %s", url, resp.Status)
    }
    doc, err := html.Parse(resp.Body)
    resp.Body.Close()
    if err != nil {
        return nil, fmt.Errorf("parsing %s as HTML: %v", url, err)
    }
    return visit(nil, doc), nil
}
```

Há quatro instruções de retorno em **findLinks**, cada qual devolvendo um par de valores. Os três primeiros **returns** fazem a função passar os erros subjacentes dos pacotes **http** e **html** a quem fez a chamada. No primeiro

caso, o erro é devolvido sem alterações; no segundo e terceiro casos, ele é expandido com informações adicionais de contexto com `fmt.Errorf` (seção 7.8). Se `findLinks` for bem-sucedido, a última instrução `return` devolve a fatia de links, sem erro.

Devemos garantir que `resp.Body` seja fechado para que os recursos de rede sejam devidamente liberados, mesmo em caso de erro. O coletor de lixo (garbage collector) de Go recicla memória não usada, mas não libera recursos não usados do sistema operacional, como arquivos e conexões de rede abertos. Eles devem ser explicitamente fechados.

O resultado da chamada de uma função que devolve múltiplos valores é uma tupla de valores. Quem chama uma função desse tipo deve atribuir explicitamente os valores a variáveis se alguma delas vai ser usada:

```
links, err := findLinks(url)
```

Para ignorar um dos valores, atribua-o ao identificador vazio:

```
links, _ := findLinks(url) // erros são ignorados
```

O resultado de uma chamada que devolve múltiplos valores pode ser devolvido a partir da chamada de uma função (que devolve múltiplos valores), como nesta que se comporta como `findLinks`, mas faz log de seu argumento:

```
func findLinksLog(url string) ([]string, error) {
    log.Printf("findLinks %s", url)
    return findLinks(url)
}
```

Uma chamada de função que devolve múltiplos valores pode aparecer como único argumento na chamada de uma função com vários parâmetros. Embora raramente seja usado em código de produção, esse recurso às vezes é conveniente durante o debugging, pois nos permite exibir todos os resultados de uma chamada com uma única instrução. As duas instruções de exibição a seguir têm o mesmo efeito.

```
log.Println(findLinks(url))
links, err := findLinks(url)
log.Println(links, err)
```

Nomes bem escolhidos podem documentar o significado dos resultados de uma função. Nomes são particularmente importantes quando uma função devolve vários resultados de mesmo tipo, como:

```
func Size(rect image.Rectangle) (width, height int)
func Split(path string) (dir, file string)
func HourMinSec(t time.Time) (hour, minute, second int)
```

mas nem sempre é necessário nomear múltiplos resultados somente por questões de documentação. Por exemplo, a convenção determina que um resultado final **bool** indica sucesso; um resultado **error** geralmente não exige explicações.

Em uma função com resultados nomeados, os operandos de uma instrução de retorno podem ser omitidos. Isso se chama *retorno descoberto* (bare return).

```
// CountWordsAndImages faz uma requisição HTTP GET para o documento
// HTML em url e devolve o número de palavras e imagens contidas.
func CountWordsAndImages(url string) (words, images int, err error) {
    resp, err := http.Get(url)
    if err != nil {
        return
    }
    doc, err := html.Parse(resp.Body)
    resp.Body.Close()
    if err != nil {
        err = fmt.Errorf("parsing HTML: %s", err)
        return
    }
    words, images = countWordsAndImages(doc)
    return
}

func countWordsAndImages(n *html.Node) (words, images int) { /* ... */ }
```

Um retorno descoberto é uma maneira concisa de devolver cada uma das variáveis nomeadas de resultado na sequência; desse modo, na função anterior, cada instrução de retorno é equivalente a:

```
return words, images, err
```

Em funções como essa, com muitas instruções de retorno e vários resultados, retornos descobertos podem reduzir a duplicação de código,

mas raramente deixam o código mais fácil de entender. Por exemplo, à primeira vista, não é óbvio que os dois primeiros retornos são equivalentes a `return 0, 0, err` (porque as variáveis de resultado `words` e `images` são inicializadas com seus valores zero) e que a última instrução `return` é equivalente a `return words, images, nil`. Por esse motivo, retornos descobertos devem ser usados com moderação.

Exercício 5.5: Implemente `countWordsAndImages`. (Veja o exercício 4.9 para saber como separar palavras.)

Exercício 5.6: Modifique a função `corner` em `gopl.io/ch3/surface` (seção 3.2) para que use resultados nomeados e uma instrução de retorno vazia.

5.4 Erros

Algumas funções sempre são bem-sucedidas em suas tarefas. Por exemplo, `strings.Contains` e `strconv.FormatBool` têm resultados bem definidos para todos os valores de argumento possíveis e não podem falhar – exceto em cenários catastróficos e imprevisíveis, como ficar sem memória, caso em que o sintoma está longe da causa e em que há poucas esperanças de recuperação.

Outras funções são sempre bem-sucedidas, desde que suas pré-condições sejam atendidas. Por exemplo, a função `time.Date` sempre constrói um `time.Time` a partir de seus componentes – ano, mês, e assim por diante – a menos que o último argumento (o fuso horário) seja `nil`, caso em que um pânico é gerado. Esse pânico é um sinal certo de um bug no código que fez a chamada e jamais deveria ocorrer em um programa bem escrito.

Para muitas outras funções, até mesmo em um programa bem escrito o sucesso não é garantido, pois depende de fatores que estão além do controle do programador. Qualquer função que faça E/S, por exemplo, deve levar em conta a possibilidade de erro, e somente um programador ingênuo acreditaria que uma leitura ou uma escrita simples não poderiam falhar. De fato, quando as operações mais confiáveis falham inesperadamente é que mais precisamos saber o porquê.

Desse modo, erros são uma parte importante da API de um pacote ou da interface de usuário de uma aplicação, e a falha é apenas um dos vários comportamentos esperados. Essa é a abordagem de Go para tratamento de erros.

Uma função para a qual a falha é um comportamento esperado devolve um resultado adicional que, por convenção, é o último. Se a falha tiver apenas uma causa possível, o resultado será um booleano, normalmente chamado **ok**, como no exemplo a seguir de uma busca em cache que sempre será bem-sucedida, a menos que não haja uma entrada para a chave:

```
value, ok := cache.Lookup(key)
if !ok {
    // ...cache[key] não existe...
}
```

Com mais frequência, em especial para E/S, a falha pode ter várias causas para as quais quem fez a chamada precisa de uma explicação. Em casos como esses, o tipo do resultado adicional é **error**.

O tipo embutido **error** é um tipo interface. Veremos mais sobre o que isso significa e suas implicações para o tratamento de erros no capítulo 7. Por enquanto, é suficiente saber que um **error** pode ser nil ou não; que nil implica sucesso e não nil quer dizer falha; e que um **error** diferente de nil contém uma string de mensagem de erro que podemos obter chamando seu método **Error** ou que pode ser exibida chamando **fmt.Println(err)** ou **fmt.Printf("%v", err)**.

Em geral, quando uma função devolve um erro diferente de nil, seus outros resultados são indefinidos e devem ser ignorados. No entanto, algumas funções podem devolver resultados parciais em casos de erro. Por exemplo, se um erro ocorre durante a leitura de um arquivo, uma chamada a **Read** devolve o número de bytes que puderam ser lidos e um valor **error** que descreve o problema. Para ter um comportamento correto, alguns códigos que fazem a chamada podem precisar processar os dados incompletos antes de tratar o erro, portanto é importante que essas funções documentem claramente seus resultados.

A abordagem de Go a diferencia de várias outras linguagens em que as falhas são informadas usando *exceções* no lugar de valores comuns. Embora Go tenha uma espécie de sistema de exceções, como veremos na seção 5.9, ele é usado apenas para informar erros realmente inesperados que indicam um bug, e não erros rotineiros que um programa robusto deva esperar.

O motivo para esse design é que as exceções tendem a misturar a descrição de um erro com o controle de fluxo necessário para tratá-lo, muitas vezes levando a um resultado indesejado: erros rotineiros são informados ao usuário final na forma de uma stack trace (estado da pilha de execução) incompreensível, cheia de informações sobre a estrutura do programa, mas sem um contexto inteligível sobre o que saiu errado.

Em comparação, programas Go usam sistemas comuns de controle de fluxo como `if` e `return` para responder a erros. Sem dúvida, esse estilo exige que se preste mais atenção na lógica de tratamento de erros, mas essa é exatamente a ideia.

5.4.1 Estratégias de tratamento de erros

Quando uma chamada de função devolve um erro, é responsabilidade de quem chamou verificá-lo e tomar a atitude apropriada. De acordo com a situação, pode haver algumas possibilidades. Vamos dar uma olhada em cinco delas.

A primeira opção, e também a mais comum, é *propagar* o erro, de modo que uma falha em uma sub-rotina torne-se uma falha na rotina que fez a chamada. Vimos exemplos disso na função `findLinks` na seção 5.3. Se a chamada a `http.Get` falhar, `findLinks` simplesmente devolve o erro de HTTP a quem chamou:

```
resp, err := http.Get(url)
if err != nil {
    return nil, err
}
```

Em comparação, se a chamada a `html.Parse` falhar, `findLinks` não

devolve o erro do parser HTML diretamente, pois faltam duas informações cruciais: que o erro ocorreu no parser e o URL do documento submetido ao parsing. Nesse caso, **findLinks** cria uma nova mensagem de erro que inclui ambas as informações, assim como o erro de parse subjacente:

```
doc, err := html.Parse(resp.Body)
resp.Body.Close()
if err != nil {
    return nil, fmt.Errorf("parsing %s as HTML: %v", url, err)
}
```

A função **fmt.Errorf** formata uma mensagem de erro usando **fmt.Sprintf** e devolve um novo valor de **error**. Ela é usada para criar erros descritivos ao prefixar sucessivamente informações adicionais de contexto à mensagem de erro original. Quando o erro, em última instância, for tratado pela função **main** do programa, ele deverá oferecer um encadeamento claro da causa-raiz do problema para a falha como um todo, que lembre uma investigação de acidentes da NASA:

```
Genesis: destruída no impacto: para-quedas não abriu: falha na G-switch: sensor
instalado na direção errada
```

Como as mensagens de erro com frequência são encadeadas, as strings com as mensagens não devem começar com letra maiúscula, e o uso de quebras de linha deve ser evitado. Os erros resultantes podem ser longos, mas parecerão completos quando encontrados por ferramentas como **grep**.

Ao conceber mensagens de erro, seja meticuloso, de modo que cada mensagem tenha uma descrição clara do problema, com detalhes suficientes e relevantes; além disso, seja consistente, de modo que erros devolvidos pela mesma função ou por um grupo de funções do mesmo pacote sejam semelhantes quanto à forma e possam ser tratados da mesma maneira.

Por exemplo, o pacote **os** garante que todo erro devolvido por uma operação de arquivo, como **os.Open** ou os métodos **Read**, **Write** ou **Close** de um arquivo aberto, descreva não só a natureza da falha (sem

permissão, diretório inexistente, e assim por diante), mas também o nome do arquivo; portanto, quem chamou não precisa incluir essa informação na mensagem de erro construída.

Em geral, a chamada a `f(x)` é responsável por informar a operação `f` que se tentou executar e o valor do argumento `x`, pois eles estão relacionados ao contexto do erro. Quem fez a chamada é responsável por acrescentar outras informações que tiver, mas que `f(x)` não tenha, como o URL na chamada anterior a `html.Parse`.

Vamos passar para a segunda estratégia de tratamento de erros. Para erros que representem problemas transientes ou imprevisíveis, pode ser interessante fazer uma *nova tentativa* (retry) da operação que falhou, possivelmente com um intervalo de tempo entre as tentativas, e, quem sabe, com um limite no número de tentativas ou no tempo gasto com elas antes de desistir totalmente.

gopl.io/ch5/wait

```
// WaitForServer tenta contatar o servidor de um URL.
// Tenta por um minuto usando exponential back-off1.
// Devolve um erro se todas as tentativas falharem.
func WaitForServer(url string) error {
    const timeout = 1 * time.Minute
    deadline := time.Now().Add(timeout)
    for tries := 0; time.Now().Before(deadline); tries++ {
        _, err := http.Head(url)
        if err == nil {
            return nil // sucesso
        }
        log.Printf("server not responding (%s); retrying...", err)
        time.Sleep(time.Second << uint(tries)) // exponential back-off
    }
    return fmt.Errorf("server %s failed to respond after %s", url, timeout)
}
```

Em terceiro lugar, se for impossível continuar, quem fez a chamada pode exibir o erro e interromper o programa com elegância, mas esse curso de ação em geral deve ser reservado para o pacote principal de um programa. Funções de biblioteca normalmente devem propagar erros a quem

chamou, a menos que o erro seja um sinal de inconsistência interna – isto é, um bug.

```
// (Na função main.)
if err := WaitForServer(url); err != nil {
    fmt.Fprintf(os.Stderr, "Site is down: %v\n", err)
    os.Exit(1)
}
```

Um modo mais conveniente de obter o mesmo efeito é chamar **log.Fatalf**. Como ocorre com todas as funções de **log**, por padrão, o horário e a data são prefixados na mensagem de erro.

```
if err := WaitForServer(url); err != nil {
    log.Fatalf("Site is down: %v\n", err)
}
```

O formato default é útil em um servidor que execute por bastante tempo, mas ajuda menos em uma ferramenta interativa:

```
2006/01/02 15:04:05 Site is down: no such domain: bad.gopl.io
```

Para uma saída mais atraente, podemos definir o prefixo usado pelo pacote **log** com o nome do comando e remover a exibição de data e hora:

```
log.SetPrefix("wait: ")
log.SetFlags(0)
```

Como quarta opção, em alguns casos, basta apenas fazer log do erro e, em seguida, prosseguir, talvez com funcionalidades reduzidas. Novamente, há uma opção entre usar o pacote **log**, que acrescenta o prefixo usual:

```
if err := Ping(); err != nil {
    log.Printf("ping failed: %v; networking disabled", err)
}
```

e fazer a exibição diretamente no stream do erro-padrão:

```
if err := Ping(); err != nil {
    fmt.Fprintf(os.Stderr, "ping failed: %v; networking disabled\n", err)
}
```

(Todas as funções de **log** concatenam uma quebra de linha, se ainda não houver uma presente.)

Como quinta e última opção, em casos raros, podemos ignorar totalmente um erro de forma segura:

```
dir, err := ioutil.TempDir("", "scratch")
if err != nil {
    return fmt.Errorf("failed to create temp dir: %v", err)
}

// ...usa o diretório temp...

os.RemoveAll(dir) // ignora erros; $TMPDIR é limpo periodicamente
```

A chamada a **os.RemoveAll** pode falhar, mas o programa ignora isso, porque o sistema operacional limpa periodicamente o diretório temporário. Nesse caso, descartar o erro foi intencional, mas a lógica do programa seria a mesma caso tivéssemos esquecido de tratá-lo. Cultive o hábito de considerar os erros após toda chamada de função; quando ignorar um erro deliberadamente, documente a sua intenção de forma clara.

O tratamento de erros em Go tem um ritmo particular. Após verificar um erro, normalmente a falha é tratada antes do sucesso. Se a falha fizer a função retornar, a lógica para o sucesso não é indentada em um bloco **else**, mas vem a seguir no nível mais externo. Funções tendem a exibir uma estrutura comum, com uma série de verificações iniciais para rejeitar erros, seguida da parte principal da função no final, minimamente indentada.

5.4.2 Fim de arquivo (EOF)

Geralmente, a variedade de erros que uma função pode devolver é interessante para o usuário final, mas não para a lógica do programa. Às vezes, porém, um programa deve tomar atitudes diferentes de acordo com o tipo de erro que ocorreu. Considere uma tentativa de ler n bytes de dados de um arquivo. Se n for escolhido para ser o tamanho do arquivo, qualquer erro representa uma falha. Por outro lado, se quem faz a chamada tentar repetidamente ler porções de tamanho fixo até que o arquivo acabe, sua resposta a uma condição de fim de arquivo deve ser diferente se comparada a todos os demais erros. Por esse motivo, o pacote **io** garante que qualquer falha de leitura provocada por uma condição de fim de arquivo seja sempre informada por um erro especial, **io.EOF**,

definido da seguinte maneira:

```
package io

import "errors"
// EOF é o erro retornado por Read quando não houver mais dados de
// entrada disponíveis.
var EOF = errors.New("EOF")
```

Quem faz a chamada pode identificar essa condição usando uma comparação simples, como no loop a seguir, que lê runas da entrada-padrão. (O programa **charcount** da seção 4.3 oferece um exemplo mais completo.)

```
in := bufio.NewReader(os.Stdin)
for {
    r, _, err := in.ReadRune()
    if err == io.EOF {
        break // leitura terminada
    }
    if err != nil {
        return fmt.Errorf("read failed: %v", err)
    }
    // ...usa r...
}
```

Em uma condição de fim de arquivo, como não há informações a serem dadas além de sua ocorrência, **io.EOF** tem uma mensagem de erro fixa, "EOF". Para outros erros, pode ser necessário informar tanto a qualidade quanto a quantidade do erro, por assim dizer, portanto um valor de erro fixo não será conveniente. Na seção 7.11, apresentaremos uma maneira mais sistemática de diferenciar determinados valores de erros de outros.

5.5 Valores função

Funções são *valores de primeira classe* (first-class values) em Go: como outros valores, valores função têm tipos e podem ser atribuídos a variáveis, ou passados para funções, ou serem devolvidos por elas. Um valor função pode ser chamado como qualquer outra função. Por exemplo:

```
func square(n int) int    { return n * n }
```

```

func negative(n int) int { return -n }
func product(m, n int) int { return m * n }

f := square
fmt.Println(f(3)) // "9"

f = negative
fmt.Println(f(3)) // "-3"
fmt.Printf("%T\n", f) // "func(int) int"

f = product // erro de compilação: can't assign func(int, int) int
             // to func(int) int

```

O valor zero de um tipo função é **nil**. Chamar um valor função nil gera um pânico:

```

var f func(int) int
f(3) // pânico: chamada de função nil

```

Valores função podem ser comparados com **nil**:

```

var f func(int) int
if f != nil {
    f(3)
}

```

mas não são comparáveis, ou seja, não podem ser comparados uns com os outros nem usados como chaves em um mapa.

Valores função permitem parametrizar nossas funções não só com dados, mas em comportamento também. As bibliotecas-padrão contêm vários exemplos. Por exemplo, **strings.Map** aplica uma função a cada caractere de uma string, unindo os resultados para compor outra string.

```

func add1(r rune) rune { return r + 1 }

fmt.Println(strings.Map(add1, "HAL-9000")) // "IBM.:111"
fmt.Println(strings.Map(add1, "VMS"))      // "WNT"
fmt.Println(strings.Map(add1, "Admix"))    // "Benjy"

```

A função **findLinks** da seção 5.2 usa uma função auxiliar **visit** para visitar todos os nós de um documento HTML e aplicar uma ação a cada um. Usando um valor função, podemos separar a lógica para percorrer a árvore da lógica da ação a ser aplicada a cada nó, permitindo reutilizar o processo de percorrer a árvore com diferentes ações.

gopl.io/ch5/outline2

```
// forEachNode chama as funções pre(x) e post(x) para cada nó
// x da árvore cuja raiz é n. Ambas as funções são opcionais.
// pre é chamada antes de os filhos serem visitados (pré-ordem) e
// post é chamada depois (pós-ordem).
func forEachNode(n *html.Node, pre, post func(n *html.Node)) {
    if pre != nil {
        pre(n)
    }

    for c := n.FirstChild; c != nil; c = c.NextSibling {
        forEachNode(c, pre, post)
    }

    if post != nil {
        post(n)
    }
}
```

A função **forEachNode** aceita dois argumentos de função, um para ser chamado antes de os filhos de um nó serem visitados e outro para ser chamado depois. Essa organização confere grande flexibilidade a quem faz a chamada. Por exemplo, as funções **startElement** e **endElement** exibem as tags de início e de fim de um elemento HTML como **...**:

```
var depth int

func startElement(n *html.Node) {
    if n.Type == html.ElementNode {
        fmt.Printf("%*s<%s>\n", depth*2, "", n.Data)
        depth++
    }
}

func endElement(n *html.Node) {
    if n.Type == html.ElementNode {
        depth--
        fmt.Printf("%*s</%s>\n", depth*2, "", n.Data)
    }
}
```

As funções também indentam a saída usando outro truque de **fmt.Printf**. O advérbio ***** em **%*s** exibe uma string preenchida com um número variável de espaços. A largura e a string são fornecidas pelos argumentos **depth*2** e **""**.

Se chamarmos **forEachNode** em um documento HTML deste modo:

```
forEachNode(doc, startElement, endElement)
```

vamos obter uma variação mais sofisticada da saída, quando comparada ao nosso programa **outline** anterior:

```
$ go build gopl.io/ch5/outline2
```

```
$ ./outline2 http://gopl.io
```

```
<html>
  <head>
    <meta>
  </meta>
    <title>
  </title>
    <style>
  </style>
  </head>
  <body>
    <table>
      <tbody>
        <tr>
          <td>
            <a>
              <img>
            </img>
          ...
```

Exercício 5.7: Desenvolva **startElement** e **endElement** em um pretty-printer HTML genérico. Exiba nós de comentários, nós de texto e os atributos de cada elemento (****). Use formas compactas como **** no lugar de **** quando um elemento não tiver filhos. Escreva um teste para garantir que seja possível fazer parse da saída com sucesso. (Veja o capítulo 11.)

Exercício 5.8: Modifique **forEachNode** para que as funções **pre** e **post** devolvam um resultado booleano indicando se devem continuar o percurso. Use-a para escrever uma função **ElementByID** com a assinatura a seguir para encontrar o primeiro elemento HTML com o atributo **id** especificado. A função deve parar de percorrer os elementos assim que uma correspondência for feita.

```
func ElementByID(doc *html.Node, id string) *html.Node
```

Exercício 5.9: Escreva uma função `expand(s string, f func(string) string) string` que substitua cada substring “\$foo” em `s` pelo texto devolvido por `f("foo")`.

5.6 Funções anônimas

Funções nomeadas podem ser declaradas somente no nível de pacote, mas podemos usar uma *função literal* para representar um valor função em qualquer expressão. Uma função literal é escrita como uma declaração de função, mas sem um nome após a palavra reservada `func`. É uma expressão, e seu valor chama-se *função anônima*.

Funções literais permitem definir uma função no ponto em que ela é usada. Como exemplo, a chamada anterior a `strings.Map` pode ser reescrita como:

```
strings.Map(func(r rune) rune { return r + 1 }, "HAL-9000")
```

Mais importante ainda, funções definidas dessa maneira têm acesso a todo o ambiente léxico, portanto a função interna pode fazer referência a variáveis da função que a engloba, como mostra o exemplo a seguir:

gopl.io/ch5/squares

```
// squares devolve uma função que devolve
// o próximo quadrado perfeito sempre que ela é chamada.
func squares() func() int {
    var x int
    return func() int {
        x++
        return x * x
    }
}

func main() {
    f := squares()
    fmt.Println(f()) // "1"
    fmt.Println(f()) // "4"
    fmt.Println(f()) // "9"
    fmt.Println(f()) // "16"
}
```


A função **squares** devolve outra função, que é do tipo `func() int`. Uma chamada a **squares** cria uma variável local **x** e devolve uma função anônima que, sempre que é chamada, incrementa **x** e devolve seu quadrado. Uma segunda chamada a **squares** cria uma segunda variável **x** e devolve uma nova função anônima que incrementa essa variável.

O exemplo com **squares** demonstra que valores função não são apenas código, mas podem ter estados. A função anônima interna pode acessar e atualizar as variáveis locais da função **squares** que a engloba. Essas referências a variáveis ocultas são o motivo pelo qual classificamos funções como tipos referência e pelo qual valores função não são comparáveis. Valores função como esses são implementados usando uma técnica chamada *closures*, e programadores Go com frequência usam esse termo para valores função.

Vemos novamente aqui um exemplo em que o tempo de vida de uma variável não é determinado pelo seu escopo: a variável **x** existe após **squares** ter retornado em **main**, apesar de **x** estar oculto em **f**.

Como uma espécie de exemplo acadêmico de funções anônimas, considere o problema de processar uma sequência de disciplinas de ciência da computação em que os pré-requisitos de cada uma sejam satisfeitos. Os pré-requisitos são dados pela tabela **prereqs** a seguir, que é um mapeamento de cada disciplina para a lista das que devem ser concluídas antes dela.

gopl.io/ch5/toposort

```
// prereqs mapeia disciplinas de ciência da computação aos seus pré-requisitos.
var prereqs = map[string][]string{
    "algorithms": {"data structures"},
    "calculus":   {"linear algebra"},

    "compilers": {
        "data structures",
        "formal languages",
        "computer organization",
    },

    "data structures": {"discrete math"},
    "databases":      {"data structures"},
}
```

```

    "discrete math":      {"intro to programming"},
    "formal languages":   {"discrete math"},
    "networks":           {"operating systems"},
    "operating systems":  {"data structures", "computer organization"},
    "programming languages": {"data structures", "computer organization"},
}

```

Esse tipo de problema é conhecido como ordenação topológica (topological sort). Conceitualmente, as informações de pré-requisitos formam um grafo dirigido, com um nó para cada disciplina e arestas de cada disciplina para aquelas que dependem dela. O grafo é acíclico: não há nenhum caminho que parta de uma disciplina e retorne a ela. Podemos calcular uma sequência válida usando busca em profundidade (depth-first search)² pelo grafo com o código a seguir:

```

func main() {
    for i, course := range topoSort(prereqs) {
        fmt.Printf("%d:\t%s\n", i+1, course)
    }
}

func topoSort(m map[string][]string) []string {
    var order []string
    seen := make(map[string]bool)
    var visitAll func(items []string)
    visitAll = func(items []string) {
        for _, item := range items {
            if !seen[item] {
                seen[item] = true
                visitAll(m[item])
                order = append(order, item)
            }
        }
    }

    var keys []string
    for key := range m {
        keys = append(keys, key)
    }

    sort.Strings(keys)
    visitAll(keys)
    return order
}

```

Quando uma função anônima exige recursão, como nesse exemplo, devemos inicialmente declarar uma variável e então atribuir a função anônima a ela. Se esses dois passos tivessem sido combinados na declaração, a função literal não estaria no escopo da variável **visitAll**, portanto não haveria nenhuma maneira de chamá-la recursivamente:

```
visitAll := func(items []string) {  
    // ...  
    visitAll(m[item]) // erro de compilação: visitAll não está definido  
    // ...  
}
```

A saída do programa **toposort** está apresentada a seguir. Ela é determinística, uma propriedade muitas vezes desejável, que nem sempre vem gratuitamente. Nesse caso, os valores do mapa **prereqs** são fatias, e não outros mapas. Assim, sua ordem de iteração é determinística, e ordenamos as chaves de **prereqs** antes de fazer as chamadas iniciais a **visitAll**.

```
1:    intro to programming  
2:    discrete math  
3:    data structures  
4:    algorithms  
5:    linear algebra  
6:    calculus  
7:    formal languages  
8:    computer organization  
9:    compilers  
10:   databases  
11:   operating systems  
12:   networks  
13:   programming languages
```

Vamos voltar ao exemplo com **findLinks**. Transferimos a função de extração de links **links.Extract** para seu próprio pacote, pois ela será usada novamente no capítulo 8. Substituímos a função **visit** por uma função anônima que concatena itens à fatia **links** diretamente e usamos **forEachNode** para tratar o percurso. Como **Extract** precisa somente da função **pre**, **nil** é passado como o argumento **post**.

gopl.io/ch5/links

```

// O pacote links oferece uma função para extração de links.
package links

import (
    "fmt"
    "net/http"
    "golang.org/x/net/html"
)

// Extract faz uma requisição HTTP GET ao URL especificado, faz parse
// da resposta como HTML e devolve os links do documento HTML.
func Extract(url string) ([]string, error) {
    resp, err := http.Get(url)
    if err != nil {
        return nil, err
    }
    if resp.StatusCode != http.StatusOK {
        resp.Body.Close()
        return nil, fmt.Errorf("getting %s: %s", url, resp.Status)
    }

    doc, err := html.Parse(resp.Body)
    resp.Body.Close()
    if err != nil {
        return nil, fmt.Errorf("parsing %s as HTML: %v", url, err)
    }

    var links []string
    visitNode := func(n *html.Node) {
        if n.Type == html.ElementNode && n.Data == "a" {
            for _, a := range n.Attr {
                if a.Key != "href" {
                    continue
                }
                link, err := resp.Request.URL.Parse(a.Val)
                if err != nil {
                    continue // ignora URLs ruins
                }
                links = append(links, link.String())
            }
        }
    }
    forEachNode(doc, visitNode, nil)
    return links, nil
}

```

Em vez de concatenar o valor puro do atributo **href** à fatia **links**, essa versão faz parse desse dado como um URL relativo ao URL base do documento, **resp.Request.URL**. O **link** resultante está na forma absoluta, adequada para uso em uma chamada a **http.Get**.

Fazer crawling da web, em sua essência, é uma questão de percorrer um grafo. O exemplo com **topoSort** mostrou uma travessia em profundidade; em nosso web crawler, usaremos a travessia em largura (breadth-first traversal)³, pelo menos inicialmente. No capítulo 8, exploraremos a travessia concorrente (concurrent traversal).

A função a seguir encapsula a essência de uma travessia em largura. Quem faz a chamada fornece uma lista inicial **worklist** com itens a visitar e um valor função **f** a ser chamado para cada um deles. Cada item é identificado por uma string. A função **f** devolve uma lista de novos itens a serem concatenados na lista de trabalho (**worklist**). A função **breadthFirst** retorna quando todos os itens tiverem sido visitados. Ela mantém um conjunto de strings para garantir que nenhum item seja visitado duas vezes.

gopl.io/ch5/findlinks3

```
// breadthFirst chama f para cada item em worklist.
// Todo item devolvido por f é adicionado à worklist.
// f é chamada no máximo uma vez para cada item.
func breadthFirst(f func(item string) []string, worklist []string) {
    seen := make(map[string]bool)
    for len(worklist) > 0 {
        items := worklist
        worklist = nil
        for _, item := range items {
            if !seen[item] {
                seen[item] = true
                worklist = append(worklist, f(item)...)
            }
        }
    }
}
```

Conforme explicamos no capítulo 4, o argumento “**f(item)...**” faz

todos os itens da lista devolvida por **f** serem concatenados na **worklist**.

Em nosso crawler, os itens são URLs. A função **crawl** que forneceremos a **breadthFirst** exhibe o URL, extrai seus links e os devolve para que eles também sejam visitados.

```
func crawl(url string) []string {
    fmt.Println(url)
    list, err := links.Extract(url)
    if err != nil {
        log.Print(err)
    }
    return list
}
```

Para iniciar o crawler, usaremos os argumentos de linha de comando como os URLs iniciais.

```
func main() {
    // Faz crawling da web em largura (breadth-first),
    // começando com os argumentos da linha de comando.
    breadthFirst(crawl, os.Args[1:])
}
```

Vamos fazer crawling da web a partir de **https://golang.org**. Eis alguns dos links resultantes:

```
$ go build gopl.io/ch5/findlinks3
$ ./findlinks3 https://golang.org
https://golang.org/
https://golang.org/doc/
https://golang.org/pkg/
https://golang.org/project/
https://code.google.com/p/go-tour/
https://golang.org/doc/code.html
https://www.youtube.com/watch?v=XCsl89YtqCs
http://research.swtch.com/gotour
https://vimeo.com/53221560
...
```

O processo termina quando todas as páginas web acessíveis forem percorridas ou a memória do computador se esgotar.

Exercício 5.10: Reescreva **topoSort** para que use mapas no lugar de fatias e elimine a ordenação inicial. Verifique se os resultados (embora não sejam

determinísticos) são ordenações topológicas válidas.

Exercício 5.11: O instrutor da disciplina de álgebra linear decidiu que cálculo agora é um pré-requisito. Estenda a função `topoSort` para que informe ciclos.

Exercício 5.12: As funções `startElement` e `endElement` em gopl.io/ch5/outline2 (seção 5.5) compartilham uma variável global `depth`. Transforme-as em funções anônimas que compartilham uma variável local à função `outline`.

Exercício 5.13: Modifique `crawl` para que faça cópias locais das páginas que encontrar, criando diretórios conforme for necessário. Não faça cópias de páginas provenientes de um domínio diferente. Por exemplo, se a página original vier de golang.org, salve todos os arquivos que estiverem aí, mas exclua aqueles de vimeo.com.

Exercício 5.14: Use a função `breadthFirst` para explorar uma estrutura diferente. Por exemplo, você poderia usar as dependências de disciplinas do exemplo com `topoSort` (um grafo direcionado), a hierarquia do sistema de arquivos de seu computador (uma árvore) ou uma lista de rotas de ônibus ou de metrô baixada do site da prefeitura de sua cidade (um grafo não direcionado).

5.6.1 Cuidado: captura de variáveis de iteração

Nesta seção, vamos analisar uma armadilha das regras de escopo léxico de Go que pode causar resultados surpreendentes. Recomendamos que você compreenda o problema antes de continuar, pois até mesmo programadores experientes podem cair nessa armadilha.

Considere um programa que deva criar um conjunto de diretórios e, posteriormente, removê-los. Podemos usar uma fatia com valores de funções para armazenar as operações de limpeza. (Por questões de concisão, omitimos todo o tratamento de erros deste exemplo.)

```
var rmdirs []func()
for _, d := range tempDirs() {
    dir := d                // NOTA: necessário!
```

```

    os.MkdirAll(dir, 0755) // cria diretórios-pais também
    rmdirs = append(rmdirs, func() {
        os.RemoveAll(dir)
    })
}
// ...executa algumas tarefas...
for _, rmdir := range rmdirs {
    rmdir() // limpa
}

```

Você pode estar se perguntando por que atribuímos a variável de loop **d** a uma nova variável local **dir** no corpo do loop, em vez de simplesmente chamar a variável de loop de **dir**, como nesta variante sutilmente incorreta:

```

var rmdirs []func()
for _, dir := range tempDirs() {
    os.MkdirAll(dir, 0755)
    rmdirs = append(rmdirs, func() {
        os.RemoveAll(dir) // NOTA: incorreto!
    })
}

```

O motivo é uma consequência das regras de escopo para variáveis de loop. No programa imediatamente anterior, o loop **for** introduz um novo bloco léxico em que a variável **dir** é declarada. Todos os valores função criados por esse loop “capturam” e compartilham a mesma variável – um local de armazenamento endereçável, e não seu valor naquele instante em particular. O valor de **dir** é atualizado em iterações sucessivas, portanto, quando as funções de limpeza são chamadas, a variável **dir** foi atualizada diversas vezes pelo loop **for** agora concluído. Desse modo, **dir** armazena o valor da última iteração e, conseqüentemente, todas as chamadas a **os.RemoveAll** tentarão remover o mesmo diretório.

Com frequência a variável interna introduzida para contornar esse problema – **dir** em nosso exemplo – recebe o mesmo nome da variável externa da qual ela é uma cópia, resultando em declarações de variáveis de aparência estranha, porém fundamentais, como esta:

```

for _, dir := range tempDirs() {

```



```

    dir := dir // declara o dir interno, inicializado com o dir externo
    // ...
}

```

O risco não é exclusivo de loops **for** baseados em **range**. O loop no exemplo a seguir sofre do mesmo problema por causa da captura indesejada da variável de índice **i**.

```

var rmdirs []func()
dirs := tempDirs()
for i := 0; i < len(dirs); i++ {
    os.MkdirAll(dirs[i], 0755) // OK
    rmdirs = append(rmdirs, func() {
        os.RemoveAll(dirs[i]) // NOTA: incorreto!
    })
}

```

O problema da captura de variáveis de iteração ocorre com mais frequência quando usamos a instrução **go** (capítulo 8) ou com **defer** (que veremos em breve), pois ambas podem atrasar a execução de um valor função até o loop ser concluído. Mas o problema não é inerente a **go** ou **defer**.

5.7 Funções variádicas

Uma *função variádica* (variadic function) é uma função que pode ser chamada com um número variável de argumentos. Os exemplos mais familiares são **fmt.Printf** e suas variantes. **Printf** exige um argumento fixo no início e em seguida aceita qualquer quantidade de argumentos subsequentes.

Para declarar uma função variádica, o tipo do último parâmetro é precedido por reticências, “...”, que indicam que a função pode ser chamada com qualquer quantidade de argumentos desse tipo.

gopl.io/ch5/sum

```

func sum(vals ...int) int {
    total := 0
    for _, val := range vals {
        total += val
    }
}

```

```
    return total
}
```

A função **sum** anterior devolve a soma de zero ou mais argumentos **int**. No corpo da função, o tipo de **vals** é uma fatia **[]int**. Quando **sum** é chamada, qualquer quantidade de valores pode ser fornecida para seu parâmetro **vals**.

```
fmt.Println(sum())           // "0"
fmt.Println(sum(3))         // "3"
fmt.Println(sum(1, 2, 3, 4)) // "10"
```

Implicitamente, quem faz a chamada aloca um array, copia os argumentos nele e passa uma fatia do array todo para a função. Desse modo, a última chamada se comporta do mesmo modo que a chamada a seguir, que mostra como chamar uma função variádica quando os argumentos já estão em uma fatia: colocando reticências após o último argumento.

```
values := []int{1, 2, 3, 4}
fmt.Println(sum(values...)) // "10"
```

Embora o parâmetro **...int** comporte-se como uma fatia no corpo da função, o tipo de uma função variádica é diferente do tipo de uma função com um parâmetro comum de fatia.

```
func f(...int) {}
func g([]int) {}

fmt.Printf("%T\n", f) // "func(...int)"
fmt.Printf("%T\n", g) // "func([]int)"
```

Funções variádicas frequentemente são usadas para formatação de strings. A função **errorf** a seguir monta uma mensagem de erro formatada com um número de linha no início. O sufixo **f** é uma convenção de nomenclatura amplamente seguida para funções variádicas que aceitam uma string de formatação no estilo de **Printf**.

```
func errorf(linenum int, format string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, "Line %d: ", linenum)
    fmt.Fprintf(os.Stderr, format, args...)
    fmt.Fprintln(os.Stderr)
}

linenum, name := 12, "count"
errorf(linenum, "undefined: %s", name) // "Line 12: undefined: count"
```

O tipo `interface{}` quer dizer que essa função pode aceitar qualquer valor como seus argumentos finais, como explicaremos no capítulo 7.

Exercício 5.15: Escreva funções variádicas `max` e `min`, análogas a `sum`. O que essas funções devem fazer quando forem chamadas sem argumentos? Escreva variantes que exijam pelo menos um argumento.

Exercício 5.16: Escreva uma versão variádica de `strings.Join`.

Exercício 5.17: Escreva uma função variádica `ElementsByTagName` que, dada uma árvore de nós HTML e zero ou mais nomes, devolva todos os elementos que correspondam a um desses nomes. A seguir, apresentamos duas chamadas de exemplo:

```
func ElementsByTagName(doc *html.Node, name ...string) []*html.Node

images := ElementsByTagName(doc, "img")
headings := ElementsByTagName(doc, "h1", "h2", "h3", "h4")
```

5.8 Chamadas de função adiadas

Nossos exemplos de `findLinks` usavam a saída de `http.Get` como entrada para `html.Parse`. Isso funciona bem se o conteúdo do URL solicitado for realmente HTML, mas muitas páginas contêm imagens, texto simples e outros formatos de arquivo. Fornecer esses arquivos a um parser HTML pode provocar efeitos indesejados.

O programa a seguir busca um documento HTML e exibe seu título. A função `title` inspeciona o cabeçalho `Content-Type` da resposta do servidor e devolve um erro se o documento não for HTML.

gopl.io/ch5/title1

```
func title(url string) error {
    resp, err := http.Get(url)
    if err != nil {
        return err
    }

    // Verifica se Content-Type é HTML (por exemplo, "text/html;
    // charset=utf-8").
    ct := resp.Header.Get("Content-Type")
    if ct != "text/html" && !strings.HasPrefix(ct, "text/html;") {
```

```

    resp.Body.Close()
    return fmt.Errorf("%s has type %s, not text/html", url, ct)
}

doc, err := html.Parse(resp.Body)
resp.Body.Close()
if err != nil {
    return fmt.Errorf("parsing %s as HTML: %v", url, err)
}

visitNode := func(n *html.Node) {
    if n.Type == html.ElementNode && n.Data == "title" &&
        n.FirstChild != nil {
        fmt.Println(n.FirstChild.Data)
    }
}
forEachNode(doc, visitNode, nil)
return nil
}

```

Eis uma sessão típica, levemente editada para adequá-la:

```

$ go build gopl.io/ch5/title1
$ ./title1 http://gopl.io
The Go Programming Language
$ ./title1 https://golang.org/doc/effective_go.html
Effective Go - The Go Programming Language
$ ./title1 https://golang.org/doc/gopher/frontpage.png
title: https://golang.org/doc/gopher/frontpage.png
    has type image/png, not text/html

```

Observe a chamada duplicada a **resp.Body.Close()**, que garante que **title** feche a conexão de rede em todos os caminhos de execução, incluindo falhas. À medida que as funções se tornam mais complexas e precisam tratar mais erros, esse tipo de duplicação na lógica de limpeza pode se tornar um problema para a manutenção. Vejamos como o inovador mecanismo de **defer** da linguagem Go simplifica o problema.

Do ponto de vista sintático, uma instrução **defer** é uma chamada comum de função ou de método prefixada pela palavra reservada **defer**. A função e as expressões para os argumentos são avaliadas quando a instrução é executada, mas a chamada propriamente dita é *adiada* (deferred) até que a função que contém a instrução **defer** tenha

terminado, seja normalmente, executando uma instrução de retorno ou atingindo o final, seja anormalmente, por causa de um pânico. Qualquer quantidade de chamadas podem ser adiadas; elas são executadas na ordem inversa com que foram adiadas.

Uma instrução **defer** é usada com frequência em operações aos pares, como abrir e fechar, conectar e desconectar, travar e destravar (lock/unlock), para garantir que os recursos sejam liberados em todos os casos, independentemente da complexidade do controle de fluxo. O lugar correto para uma instrução **defer** que libera um recurso é logo após este ter sido obtido com sucesso. Na função **title** a seguir, uma única chamada adiada substitui as duas chamadas anteriores a **resp.Body.Close()**:

gopl.io/ch5/title2

```
func title(url string) error {
    resp, err := http.Get(url)
    if err != nil {
        return err
    }
    defer resp.Body.Close()

    ct := resp.Header.Get("Content-Type")
    if ct != "text/html" && !strings.HasPrefix(ct, "text/html;") {
        return fmt.Errorf("%s has type %s, not text/html", url, ct)
    }

    doc, err := html.Parse(resp.Body)
    if err != nil {
        return fmt.Errorf("parsing %s as HTML: %v", url, err)
    }

    // ...exibe o elemento title de doc...

    return nil
}
```

O mesmo padrão pode ser usado para outros recursos além de conexões de rede, por exemplo, para fechar um arquivo aberto:

io/ioutil

```
package ioutil
```

```
func ReadFile(filename string) ([]byte, error) {
    f, err := os.Open(filename)
    if err != nil {
        return nil, err
    }
    defer f.Close()
    return ReadAll(f)
}
```

ou para destravar um mutex (seção 9.2):

```
var mu sync.Mutex
var m = make(map[string]int)
func lookup(key string) int {
    mu.Lock()
    defer mu.Unlock()
    return m[key]
}
```

A instrução **defer** também pode ser usada para combinar um par de ações “na entrada” (on entry) e “na saída” (on exit) na depuração de uma função complexa. A função **bigSlowOperation** a seguir chama **trace** imediatamente, que executa a ação “na entrada” e, então, devolve um valor função que, quando chamado, executa a ação “na saída” correspondente. Ao adiar uma chamada à função devolvida dessa maneira, podemos aparelhar o ponto de entrada e todos os pontos de saída de uma função em uma única instrução e até mesmo passar valores, como o instante **start**, entre as duas ações. Contudo não se esqueça dos parênteses finais na instrução **defer**, ou a ação “na entrada” ocorrerá na saída e a ação “na saída” jamais será executada!

gopl.io/ch5/trace

```
func bigSlowOperation() {
    defer trace("bigSlowOperation")() // não se esqueça dos parênteses extras
    // ...muitas tarefas...
    time.Sleep(10 * time.Second) // simula uma operação demorada dormindo
}

func trace(msg string) func() {
    start := time.Now()
    log.Printf("enter %s", msg)
    return func() { log.Printf("exit %s (%s)", msg, time.Since(start)) }
```

```
}
```

Sempre que **bigSlowOperation** é chamada, ela faz log de sua entrada e de sua saída e do tempo decorrido entre elas. (Usamos **time.Sleep** para simular uma operação demorada.)

```
$ go build gopl.io/ch5/trace
$ ./trace
2015/11/18 09:53:26 enter bigSlowOperation
2015/11/18 09:53:36 exit bigSlowOperation (10.000589217s)
```

Funções adiadas executam *após* as instruções de retorno terem atualizado as variáveis de resultado da função. Como uma função anônima pode acessar as variáveis da função que a engloba, incluindo os resultados nomeados, uma função anônima adiada pode observar os resultados da função.

Considere a função **double**:

```
func double(x int) int {
    return x + x
}
```

Ao nomear sua variável de resultado e acrescentar uma instrução **defer**, podemos fazer a função exibir seus argumentos e os resultados sempre que ela for chamada.

```
func double(x int) (result int) {
    defer func() { fmt.Printf("double(%d) = %d\n", x, result) }()
    return x + x
}

_ = double(4)
// Saída:
// "double(4) = 8"
```

Esse truque é um exagero para uma função tão simples quanto **double**, mas pode ser útil em funções com muitas instruções de retorno.

Uma função anônima adiada pode até mesmo alterar os valores que a função que a engloba devolve a quem fez a chamada:

```
func triple(x int) (result int) {
    defer func() { result += x }()
    return double(x)
}
```

```
fmt.Println(triple(4)) // "12"
```

Como as funções adiadas não são executadas até o final da execução de uma função, uma instrução **defer** em um loop merece atenção extra. O código a seguir poderia ficar sem descritores de arquivo, pois nenhum arquivo será fechado até que todos eles tenham sido processados:

```
for _, filename := range filenames {
    f, err := os.Open(filename)
    if err != nil {
        return err
    }
    defer f.Close() //NOTA: arriscado; os descritores de arquivo podem se
esgotar
    // ...processa f...
}
```

Uma solução é transferir o corpo do loop, incluindo a instrução **defer**, para outra função que é chamada a cada iteração.

```
for _, filename := range filenames {
    if err := doFile(filename); err != nil {
        return err
    }
}

func doFile(filename string) error {
    f, err := os.Open(filename)
    if err != nil {
        return err
    }
    defer f.Close()
    // ...processa f...
}
```

O exemplo a seguir é uma versão melhorada do programa **fetch** (seção 1.5) que escreve a resposta HTTP em um arquivo local em vez de escrever na saída-padrão. O nome do arquivo é extraído do último componente do path do URL, obtido por meio da função **path.Base**.

gopl.io/ch5/fetch

```
// Fetch faz download do URL e devolve o
// nome e o tamanho do arquivo local.
func fetch(url string) (filename string, n int64, err error) {
```



```

resp, err := http.Get(url)
if err != nil {
    return "", 0, err
}
defer resp.Body.Close()

local := path.Base(resp.Request.URL.Path)
if local == "/" {
    local = "index.html"
}
f, err := os.Create(local)
if err != nil {
    return "", 0, err
}
n, err = io.Copy(f, resp.Body)
// Fecha o arquivo, mas dá preferência ao erro de Copy, se houver.
if closeErr := f.Close(); err == nil {
    err = closeErr
}
return local, n, err
}

```

A chamada adiada a **resp.Body.Close** deve ser familiar a essa altura. É tentador usar uma segunda chamada adiada para **f.Close** a fim de fechar o arquivo local, mas isso seria sutilmente incorreto, pois **os.Create** abre um arquivo para escrita, criando-o conforme necessário. Em muitos sistemas de arquivo, em especial no NFS, erros de escrita não são informados imediatamente, mas podem ser adiados até o arquivo ser fechado. Deixar de verificar o resultado da operação de fechamento pode fazer com que perdas sérias de dados passem despercebidas. No entanto, se tanto **io.Copy** quanto **f.Close** falharem, devemos dar preferência a informar o erro de **io.Copy**, pois ele ocorreu antes e é mais provável que informe a causa-raiz.

Exercício 5.18: Sem alterar seu comportamento, reescreva a função **fetch** de modo a usar **defer** para fechar o arquivo para escrita.

5.9 Pânico

O sistema de tipos de Go captura muitos erros em tempo de compilação,

mas outros, como um acesso fora dos limites de um array ou desreferenciar um ponteiro nil, exigem verificações em tempo de execução. Quando o runtime de Go identifica esses erros, um *pânico* é gerado.

Durante um pânico típico, a execução normal é interrompida, todas as chamadas de funções adiadas nessa gorrotina são executadas e o programa falha com uma mensagem de log. Essa mensagem de log inclui o *valor do pânico* (panic value), que geralmente é uma espécie de mensagem de erro e, para cada gorrotina, uma *stack trace* (estado da pilha de execução) mostrando a pilha das chamadas de funções que estavam ativas no momento do pânico. Em geral, a mensagem de log tem informações suficientes para diagnosticar a causa-raiz do problema sem executar de novo o programa, portanto ela sempre deve ser incluída em um relatório de bug sobre um programa que gera pânico.

Nem todo pânico é proveniente do runtime. A função embutida **panic** pode ser chamada diretamente; ela aceita qualquer valor como argumento. Gerar um pânico muitas vezes é o melhor a fazer quando uma situação “impossível” acontece, por exemplo, a execução alcança um caso que, do ponto de vista lógico, não poderia acontecer:

```
switch s := suit(drawCard()); s {
case "Spades":    // ...
case "Hearts":    // ...
case "Diamonds": // ...
case "Clubs":     // ...
default:
    panic(fmt.Sprintf("invalid suit %q", s)) // Curinga?
}
```

Assegurar que as pré-condições de uma função sejam válidas é uma boa prática, mas isso pode facilmente ser feito em excesso. A menos que você possa oferecer uma mensagem de erro mais informativa ou identificar um erro com antecedência, não há motivos para fazer a asserção de uma condição que será verificada pelo runtime.

```
func Reset(x *Buffer) {
    if x == nil {
        panic("x is nil") // desnecessário!
    }
}
```

```
x.elements = nil
}
```

Embora o sistema de pânico de Go lembre as exceções em outras linguagens, as situações em que o pânico é usado são bem diferentes. Como um pânico faz o programa terminar, em geral ele é usado para erros graves, por exemplo, uma inconsistência na lógica do programa. Programadores cuidadosos consideram qualquer terminação prematura como prova de bug em seus códigos. Em um programa robusto, erros “esperados” – o tipo que surge por causa de entradas incorretas, erros de configuração ou falhas de E/S – devem ser tratados com elegância; é melhor tratá-los usando valores de **error**.

Considere a função **regexp.Compile**, que compila uma expressão regular em um formato eficiente para casar padrões. Ela devolve um **error** se for chamada com um padrão malformado. Porém, verificar esse erro é desnecessário e trabalhoso se quem faz a chamada souber que uma chamada em particular não pode falhar. Em casos como esse, é razoável que quem faz a chamada trate um erro com pânico, pois se acredita que esse erro seja impossível.

Como a maioria das expressões regulares são literais no código-fonte do programa, o pacote **regexp** oferece uma função wrapper **regexp.MustCompile** que faz essa verificação:

```
package regexp

func Compile(expr string) (*Regexp, error) { /* ... */ }
func MustCompile(expr string) *Regexp {
    re, err := Compile(expr)
    if err != nil {
        panic(err)
    }
    return re
}
```

A função wrapper faz com que seja conveniente para os clientes inicializar uma variável de nível de pacote com uma expressão regular compilada, assim:

```
var httpSchemeRE = regexp.MustCompile(`^https?:`) // "http:" ou "https:"
```

É claro que **MustCompile** não deve ser chamada com valores de entrada não confiáveis. O prefixo **Must** é uma convenção de nomenclatura comum para funções desse tipo, como **template.Must** na seção 4.6.

Quando um pânico ocorre, todas as funções adiadas são executadas na ordem inversa, começando por aquelas que estão mais acima na pilha, continuando até **main**, como mostra o programa a seguir:

gopl.io/ch5/defer1

```
func main() {  
    f(3)  
}  
  
func f(x int) {  
    fmt.Printf("f(%d)\n", x+0/x) // pânico se x == 0  
    defer fmt.Printf("defer %d\n", x)  
    f(x - 1)  
}
```

Ao ser executado, o programa exibe o seguinte na saída-padrão:

```
f(3)  
f(2)  
f(1)  
defer 1  
defer 2  
defer 3
```

Um pânico ocorre na chamada a **f(0)**, fazendo as três chamadas adiadas a **fmt.Printf** serem executadas. Então, o runtime termina o programa, exibindo a mensagem de pânico e um dump da pilha na saída de erro padrão (simplificado por questões de clareza):

```
panic: runtime error: integer divide by zero  
main.f(0)  
    src/gopl.io/ch5/defer1/defer.go:14  
main.f(1)  
    src/gopl.io/ch5/defer1/defer.go:16  
main.f(2)  
    src/gopl.io/ch5/defer1/defer.go:16  
main.f(3)  
    src/gopl.io/ch5/defer1/defer.go:16  
main.main()  
    src/gopl.io/ch5/defer1/defer.go:10
```

Como veremos em breve, é possível que uma função se recupere de um pânico de modo a não terminar o programa.

Quando se trata de diagnóstico, o pacote `runtime` permite que o programador faça dump da pilha usando o mesmo sistema. Ao adiar uma chamada a `printStack` em `main`:

gopl.io/ch5/defer2

```
func main() {
    defer printStack()
    f(3)
}

func printStack() {
    var buf [4096]byte
    n := runtime.Stack(buf[:], false)
    os.Stdout.Write(buf[:n])
}
```

o texto adicional a seguir (novamente, simplificado por questões de clareza) é exibido na saída-padrão:

```
goroutine 1 [running]:
main.printStack()
    src/gopl.io/ch5/defer2/defer.go:20
main.f(0)
    src/gopl.io/ch5/defer2/defer.go:27
main.f(1)
    src/gopl.io/ch5/defer2/defer.go:29
main.f(2)
    src/gopl.io/ch5/defer2/defer.go:29
main.f(3)
    src/gopl.io/ch5/defer2/defer.go:29
main.main()
    src/gopl.io/ch5/defer2/defer.go:15
```

Leitores que têm familiaridade com exceções em outras linguagens podem ficar surpresos com o fato de `runtime.Stack` poder exibir informações sobre funções que pareçam já ter sido “desenroladas”. O sistema de pânico de Go executa as funções adiadas antes de “desenrolar” a pilha.

5.10 Recuperação

Desistir geralmente é a resposta correta para um pânico, mas nem sempre. Talvez seja possível recuperar-se de algum modo ou, pelo menos, arrumar a bagunça antes de encerrar. Por exemplo, um servidor web que encontre um problema inesperado poderia encerrar a conexão em vez de deixar o cliente pendurado e, durante o desenvolvimento, poderia informar o erro ao cliente também.

Se a função embutida **recover** for chamada em uma função adiada e a função que contém a instrução **defer** gerar pânico, **recover** finaliza o estado atual de pânico e devolve seu valor. A função que gerou pânico não continua do ponto em que parou, mas retorna normalmente. Se **recover** for chamado em qualquer outro instante, ele não terá nenhum efeito e devolverá **nil**.

Para ilustrar, considere o desenvolvimento de um parser para uma linguagem. Mesmo quando parece estar funcionando bem, dada a complexidade de sua tarefa, bugs podem continuar à espreita em casos raros obscuros. Podemos preferir que, em vez de causar uma falha, o parser transforme esses pânicos em erros comuns de parse, talvez com uma mensagem extra incentivando o usuário a gerar um relatório de falha.

```
func Parse(input string) (s *Syntax, err error) {
    defer func() {
        if p := recover(); p != nil {
            err = fmt.Errorf("internal error: %v", p)
        }
    }()
    // ...parser...
}
```

A função adiada em **Parse** recupera-se de um pânico, usando seu valor para compor uma mensagem de erro; uma versão mais sofisticada poderia incluir toda a pilha de chamadas (call stack) usando **runtime.Stack**. A função adiada então faz uma atribuição ao resultado **err**, que é devolvido a quem fez a chamada.

A recuperação indiscriminada de pânicos é uma prática duvidosa, pois o estado das variáveis de um pacote após um pânico raramente está bem

definido ou documentado. Talvez uma atualização crítica em uma estrutura de dados tenha ficado incompleta, um arquivo ou uma conexão de rede tenha sido aberto, mas não tenha sido fechado, ou uma trava (lock) tenha sido adquirida, mas não liberada. Além do mais, ao substituir uma falha, digamos, por uma linha em um arquivo de log, uma recuperação indiscriminada pode fazer com que bugs passem despercebidos.

Recuperar-se de um pânico no mesmo pacote pode ajudar a simplificar o tratamento de erros complexos ou inesperados, mas, como regra geral, você não deve tentar recuperar-se de pânicos de outros pacotes. As APIs públicas devem informar falhas como **errors**. De modo semelhante, você não deve se recuperar de um pânico que possa passar por uma função cuja manutenção não seja feita por você, por exemplo, uma callback fornecida por quem faz a chamada, pois não é possível dizer nada sobre a sua segurança.

Por exemplo, o pacote **net/http** disponibiliza um servidor web que faz o dispatch (despacho) de requisições de entrada para funções handler fornecidas pelo usuário. Em vez de deixar que um pânico em um desses handlers mate o processo, o servidor chama **recover**, exibe uma stack trace e continua servindo. Na prática, isso é conveniente, mas há o risco de vazamento (leaking) de recursos ou de deixar o handler que falhou em um estado não especificado, o que poderia levar a outros problemas.

Por todos os motivos anteriores, é mais seguro recuperar-se seletivamente, se é que a recuperação vai ser feita. Em outras palavras, recupere-se apenas de pânicos em que há intenção de se recuperar, o que deve ser raro. Essa intenção pode ser expressa no código usando um tipo distinto e não exportado para o valor do pânico e testando se o valor devolvido por **recover** tem esse tipo. (Veremos uma maneira de fazer isso no próximo exemplo.) Em caso afirmativo, informamos o pânico como um **error** comum; caso contrário, chamamos **panic** com o mesmo valor para restaurar o estado de pânico.

O exemplo a seguir é uma variação do programa **title** que informa um

erro se o documento HTML contiver vários elementos **<title>**. Em caso afirmativo, a recursão é abortada pela chamada a **panic** com um valor do tipo especial **bailout** (abandono).

gopl.io/ch5/title3

```
// soleTitle devolve o texto do primeiro elemento title não vazio
// em doc, e um error se não houver exatamente um.
func soleTitle(doc *html.Node) (title string, err error) {
    type bailout struct{}

    defer func() {
        switch p := recover(); p {
        case nil:
            // sem pânico
        case bailout{}:
            // pânico "esperado"
            err = fmt.Errorf("multiple title elements")
        default:
            panic(p) // pânico inesperado; prossegue com o pânico
        }
    }()

    // Sai da recursão se mais de um título não vazio for encontrado.
    forEachNode(doc, func(n *html.Node) {
        if n.Type == html.ElementNode && n.Data == "title" &&
            n.FirstChild != nil {
            if title != "" {
                panic(bailout{}) // vários elementos title
            }
            title = n.FirstChild.Data
        }
    }, nil)
    if title == "" {
        return "", fmt.Errorf("no title element")
    }
    return title, nil
}
```

A função adiada de tratamento de erro chama **recover**, verifica o valor do pânico e informa um erro comum se o valor for **bailout{}**. Todos os outros valores diferentes de nil indicam um pânico inesperado, caso em que a função chama **panic** com esse valor, eliminando o efeito de

recover e restaurando o estado original de pânico. (Esse exemplo, de certo modo, viola nosso conselho sobre não usar pânico para erros “esperados”, mas ilustra concisamente o funcionamento de **panic** e **recover**.)

De algumas condições, não há recuperação. Ficar sem memória, por exemplo, faz o runtime de Go encerrar o programa com um erro fatal.

Exercício 5.19: Use **panic** e **recover** para escrever uma função que não contenha nenhuma instrução **return**, embora devolva um valor diferente de zero.

-
- 1 Nota do Revisor da Tradução: exponential back-off (literalmente, recuo ou desistência exponencial) é uma técnica de programação de sistemas distribuídos na qual um cliente, em caso de falha na resposta, realiza novas tentativas, mas vai aumentando exponencialmente o tempo entre as tentativas, evitando congestionar ainda mais a rede ou um servidor que talvez já esteja sobrecarregado. Ou seja, em vez de insistir mais, o cliente vai desistindo gradualmente.
 - 2 N.T.: “Na teoria dos grafos, busca em profundidade (ou busca em profundidade-primeiro, também conhecido em inglês por Depth-First Search – DFS) é um algoritmo usado para realizar uma busca ou travessia numa árvore, estrutura de árvore ou grafo. Intuitivamente, o algoritmo começa num nó raiz (selecioneando algum nó como sendo o raiz, no caso de um grafo) e explora tanto quanto possível cada um dos seus ramos, antes de retroceder (backtracking).” (Fonte: https://pt.wikipedia.org/wiki/Busca_em_profundidade)
 - 3 N.T.: “Na teoria dos grafos, busca em largura (ou busca em amplitude, também conhecido em inglês por Breadth-First Search – BFS) é um algoritmo de busca em grafos utilizado para realizar uma busca ou travessia num grafo e estrutura de dados do tipo árvore. Intuitivamente, você começa pelo vértice raiz e explora todos os vértices vizinhos. Então, para cada um desses vértices mais próximos, exploramos os seus vértices vizinhos inexplorados e assim por diante, até que ele encontre o alvo da busca.” (Fonte: https://pt.wikipedia.org/wiki/Busca_em_largura)

6

Métodos

Desde o início dos anos 90, a Programação Orientada a Objetos (POO) tem sido o paradigma de programação dominante no mercado e na educação, e quase todas as linguagens amplamente usadas desenvolvidas desde então incluíram suporte a ela. Go não é exceção.

Embora não haja uma definição universalmente aceita de programação orientada a objetos, em nosso caso um *objeto* é simplesmente um valor ou uma variável que tem métodos, e um *método* é uma função associada a um tipo particular. Um programa orientado a objetos usa métodos para expressar as propriedades e operações de cada estrutura de dados, de modo que os clientes não precisem acessar a representação do objeto diretamente.

Em capítulos anteriores, usamos métodos da biblioteca-padrão frequentemente, como o método **Seconds** do tipo **time.Duration**:

```
const day = 24 * time.Hour
fmt.Println(day.Seconds()) // "86400"
```

e definimos nosso próprio método na seção 2.5 – um método **String** para o tipo **Celsius**:

```
func (c Celsius) String() string { return fmt.Sprintf("%g°C", c) }
```

Neste capítulo – o primeiro de dois capítulos sobre programação orientada a objetos – mostraremos como definir e usar métodos de forma eficiente. Também discutiremos dois princípios fundamentais da programação orientada a objetos: *encapsulamento* e *composição*.

6.1 Declarações de métodos

Um método é declarado com uma variante da declaração normal de

função, em que um parâmetro extra aparece antes do nome da função. O parâmetro associa a função ao tipo desse parâmetro.

Vamos escrever nosso primeiro método em um pacote simples para geometria plana:

gopl.io/ch6/geometry

```
package geometry

import "math"

type Point struct{ X, Y float64 }

// função tradicional
func Distance(p, q Point) float64 {
    return math.Hypot(q.X-p.X, q.Y-p.Y)
}

// a mesma função, mas como um método do tipo Point
func (p Point) Distance(q Point) float64 {
    return math.Hypot(q.X-p.X, q.Y-p.Y)
}
```

O parâmetro extra **p** é chamado de *receptor* (receiver) do método – um legado das primeiras linguagens orientadas a objetos que descreviam uma chamada de método como o “envio de uma mensagem a um objeto”.

Em Go, não usamos um nome especial como **this** ou **self** para o receptor; escolhemos nomes de receptores como faríamos para qualquer outro parâmetro. Como o nome do receptor será usado com frequência, sugere-se selecionar um nome curto e consistente entre os métodos. Uma opção comum é usar a primeira letra do nome do tipo, como **p** para **Point**.

Em uma chamada de método, o argumento para o receptor aparece antes do nome do método. Há um paralelismo com a declaração, em que o parâmetro do receptor vem antes do nome.

```
p := Point{1, 2}
q := Point{4, 6}
fmt.Println(Distance(p, q)) // "5", chamada de função
fmt.Println(p.Distance(q)) // "5", chamada de método
```

Não há conflito entre as duas declarações de funções chamadas **Distance**

apresentadas anteriormente. A primeira declara uma função no nível de pacote chamada **geometry.Distance**. A segunda declara um método do tipo **Point**, portanto seu nome é **Point.Distance**.

A expressão **p.Distance** é chamada *seletor*, pois seleciona o método **Distance** apropriado para o receptor **p** do tipo **Point**. Os seletores também são usados para selecionar campos de estruturas, como em **p.X**. Como métodos e campos estão no mesmo espaço de nomes, declarar um método **X** no tipo de estrutura **Point** seria ambíguo e rejeitado pelo compilador.

Pelo fato de cada tipo ter seu próprio espaço de nomes para métodos, podemos usar o nome **Distance** para outros métodos, desde que pertençam a tipos diferentes. Vamos definir um tipo **Path** que representa uma sequência de segmentos de reta e associar-lhe um método **Distance** também.

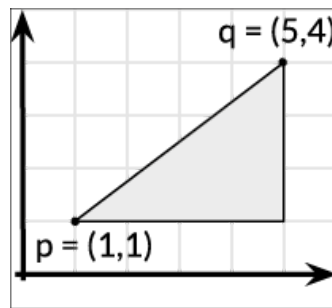
```
// Um Path é um caminho que conecta os pontos com linhas retas.
type Path []Point
// Distance devolve a distância percorrida em path.
func (path Path) Distance() float64 {
    sum := 0.0
    for i := range path {
        if i > 0 {
            sum += path[i-1].Distance(path[i])
        }
    }
    return sum
}
```

Path é um tipo fatia nomeado, e não um tipo estrutura como **Point**, mas, apesar disso, é possível definir métodos para ele. Ao permitir que métodos sejam associados a qualquer tipo, Go é diferente de muitas outras linguagens orientadas a objetos. Muitas vezes é conveniente definir comportamentos adicionais para tipos simples como números, strings, fatias, mapas e, às vezes, até mesmo funções. Os métodos podem ser declarados em qualquer tipo nomeado definido no mesmo pacote, desde que seu tipo subjacente não seja um ponteiro nem uma interface.

Os dois métodos **Distance** têm tipos diferentes; não estão relacionados

um ao outro, embora **Path.Distance** use **Point.Distance** internamente para calcular o comprimento de cada segmento que conecta pontos adjacentes.

Vamos chamar o novo método para calcular o perímetro de um triângulo retângulo:



```
perim := Path{
  {1, 1},
  {5, 1},
  {5, 4},
  {1, 1},
}
fmt.Println(perim.Distance()) // "12"
```

Nas duas chamadas anteriores aos métodos de nome **Distance**, o compilador determina qual função deve ser invocada, de acordo com o nome do método e o tipo do receptor. Na primeira chamada, **path[i-1]** é do tipo **Point**, portanto **Point.Distance** é invocada; no segundo caso, **perim** é do tipo **Path**, portanto **Path.Distance** é invocada.

Todos os métodos de um determinado tipo devem ter nomes únicos, mas tipos diferentes podem usar o mesmo nome para um método, como os métodos **Distance** para **Point** e **Path**; não há necessidade de qualificar nomes de função (por exemplo, **PathDistance**) para evitar ambiguidade. Observamos aqui a primeira vantagem de usar métodos no lugar de funções comuns: os nomes dos métodos podem ser mais compactos. A vantagem é ampliada para chamadas originadas fora do pacote, pois elas podem usar um nome mais compacto e omitir o do pacote:

```
import "gopl.io/ch6/geometry"
perim := geometry.Path[{1, 1}, {5, 1}, {5, 4}, {1, 1}]
```

```
fmt.Println(geometry.PathDistance(perim)) // "12", função independente
fmt.Println(perim.Distance())              // "12", método de geometry.Path
```

6.2 Métodos cujo receptor é um ponteiro

Como chamar uma função cria uma cópia do valor de cada argumento, se uma função precisa atualizar uma variável, ou se um argumento é tão grande a ponto de querermos evitar sua cópia, devemos passar o endereço da variável utilizando um ponteiro. Isso também vale para métodos que precisam atualizar a variável do receptor: eles são associados ao tipo ponteiro, por exemplo, ***Point**.

```
func (p *Point) ScaleBy(factor float64) {
    p.X *= factor
    p.Y *= factor
}
```

O nome desse método é **(*Point).ScaleBy**. Os parênteses são necessários, pois sem eles a expressão seria interpretada como ***(Point.ScaleBy)**.

Em um programa de verdade, a convenção determina que se algum método de **Point** tiver um receptor que seja um ponteiro, então *todos* os métodos de **Point** devem ter um receptor que seja um ponteiro, até mesmo aqueles que não precisam estritamente dele. Quebramos essa regra com **Point** para mostrar ambos os tipos de método.

Tipos nomeados (**Point**) e ponteiros para eles (***Point**) são os únicos que podem aparecer em uma declaração de receptor. Além disso, para evitar ambiguidades, declarações de métodos não são permitidas em tipos nomeados que sejam, eles mesmos, do tipo ponteiro:

```
type P *int
func (P) f() { /* ... */ } // erro compilação: tipo inválido para receptor
```

O método **(*Point).ScaleBy** pode ser chamado fornecendo-se um receptor ***Point**. Assim:

```
r := &Point{1, 2}
r.ScaleBy(2)
fmt.Println(*r) // "{2, 4}"
```

ou assim:

```
p := Point{1, 2}
pptr := &p
pptr.ScaleBy(2)
fmt.Println(p) // "{2, 4}"
```

ou, ainda, assim:

```
p := Point{1, 2}
(&p).ScaleBy(2)
fmt.Println(p) // "{2, 4}"
```

Contudo, os dois últimos casos são deslegantes. Felizmente, a linguagem nos ajuda nesse caso. Se o receptor **p** for uma *variável* do tipo **Point**, mas o método exigir um receptor ***Point**, podemos usar a forma compacta a seguir:

```
p.ScaleBy(2)
```

e o compilador executará um **&p** implícito na variável. Isso funciona somente para variáveis, incluindo campos de estrutura como **p.X** e elementos de array ou de fatia como **perim[0]**. Não podemos chamar um método de ***Point** em um receptor **Point** não endereçável, pois não há como obter o endereço de um valor temporário.

```
Point{1, 2}.ScaleBy(2) // erro de compilação: não é possível obter o
                       // endereço de um Point literal
```

Porém, *podemos* chamar um método de **Point**, como **Point.Distance**, com um receptor ***Point**, pois existe um modo de obter o valor a partir do endereço: basta carregar o valor apontado pelo receptor. O compilador insere uma operação ***** implícita. As duas chamadas de função a seguir são equivalentes:

```
pptr.Distance(q)
(*pptr).Distance(q)
```

Vamos sintetizar os três casos citados, pois são um ponto frequente de confusão. Em toda expressão válida de chamada de método, somente uma das três afirmações a seguir é verdadeira.

O argumento do receptor tem o mesmo tipo do parâmetro do receptor, por exemplo, ambos têm o tipo **T**, ou ambos têm o tipo ***T**:

```
Point{1, 2}.Distance(q) // Point
pptr.ScaleBy(2)         // *Point
```

O argumento do receptor é uma *variável* do tipo *T* e o parâmetro do receptor tem tipo **T*. O compilador implicitamente usa o endereço da variável:

```
p.ScaleBy(2) // (&p) implícito
```

O argumento do receptor é do tipo **T* e o parâmetro do receptor é do tipo *T*. O compilador implicitamente desreferencia o receptor ou, em outras palavras, carrega o valor:

```
pptr.Distance(q) // (*pptr) implícito
```

Se todos os métodos de um tipo nomeado *T* tiverem um receptor do próprio tipo *T* (e não **T*), é seguro copiar instâncias desse tipo. Chamar qualquer um de seus métodos necessariamente cria uma cópia. Por exemplo, valores de **time.Duration** são livremente copiados, inclusive como argumentos de funções. Porém, se algum método tiver um receptor que seja um ponteiro, deve-se evitar copiar instâncias de *T*, pois isso pode violar invariantes internas. Por exemplo, copiar uma instância de **bytes.Buffer** fará com que o original e a cópia sejam apelidos (*aliases*, seção 2.3.2) do mesmo array de bytes subjacente. Chamadas a métodos subsequentes terão efeitos imprevisíveis.

6.2.1 Nil é um valor válido de receptor

Assim como algumas funções permitem ponteiros nil como argumentos, isso também vale para alguns métodos quanto a seus receptores, em especial se **nil** for um valor zero significativo do tipo, como ocorre com mapas e fatias. Nesta lista ligada simples de inteiros, **nil** representa a lista vazia:

```
// Uma IntList é uma lista ligada de inteiros.
// Uma *IntList nil representa a lista vazia.
type IntList struct {
    Value int
    Tail *IntList
}
// Sum devolve a soma dos elementos da lista.
```



```
func (list *IntList) Sum() int {
    if list == nil {
        return 0
    }
    return list.Value + list.Tail.Sum()
}
```

Ao definir um tipo cujos métodos permitam `nil` como um valor de receptor, vale a pena destacar isso de forma explícita em seu comentário de documentação, como fizemos anteriormente.

A seguir, apresentamos parte da definição do tipo `Values` do pacote `net/url`:

[net/url](#)

```
package url

// Values mapeia uma chave de string a uma lista de valores.
type Values map[string][]string

// Get devolve o primeiro valor associado à chave especificada
// ou "" se não houver.
func (v Values) Get(key string) string {
    if vs := v[key]; len(vs) > 0 {
        return vs[0]
    }
    return ""
}

// Add adiciona o valor à chave.
// A concatenação é feita com eventuais valores existentes associados à chave.
func (v Values) Add(key, value string) {
    v[key] = append(v[key], value)
}
```

Esse código expõe a representação como um mapa, mas também oferece métodos para simplificar o acesso a este, cujos valores são fatias de strings – é um *multimapa* (*multimap*). Os clientes podem usar seus operadores intrínsecos (`make`, fatias literais, `m[key]`, e assim por diante) ou seus métodos, ou ambos, conforme preferirem:

[gopl.io/ch6/urlvalues](#)

```
m := url.Values{"lang": {"en"}} // construção direta
m.Add("item", "1")
```

```

m.Add("item", "2")

fmt.Println(m.Get("lang")) // "en"
fmt.Println(m.Get("q"))    // ""
fmt.Println(m.Get("item")) // "1" (primeiro valor)
fmt.Println(m["item"])     // "[1 2]" (acesso direto ao mapa)

m = nil
fmt.Println(m.Get("item")) // ""
m.Add("item", "3")         // pânico: atribuição a uma entrada em um mapa nil

```

Na última chamada a **Get**, o receptor `nil` comporta-se como um mapa vazio. Poderíamos, do mesmo modo, tê-la escrito como `Values(nil).Get("item")`, mas `nil.Get("item")` não compilará porque o tipo de `nil` não foi determinado. Em comparação, a última chamada a **Add** gera pânico, pois ela tenta atualizar um mapa `nil`.

Como `url.Values` é um tipo mapa, e um mapa refere-se a seus pares chave/valor indiretamente, quaisquer atualização e remoção que `url.Values.Add` fizer nos elementos do mapa serão visíveis a quem fizer a chamada. No entanto, como ocorre com funções comuns, qualquer mudança que um método fizer à própria referência, como defini-la para `nil` ou fazê-la referenciar uma estrutura de dados diferente do tipo mapa, não se refletirá em quem chamou.

6.3 Composto tipos por meio de inclusão de estruturas

Considere o tipo `ColoredPoint`:

gopl.io/ch6/coloredpoint

```

import "image/color"

type Point struct{ X, Y float64 }

type ColoredPoint struct {
    Point
    Color color.RGBA
}

```

Poderíamos ter definido `ColoredPoint` como uma estrutura de três campos, mas, em vez disso, *incluímos* um `Point` para fornecer os campos `X` e `Y`. Como vimos na seção 4.4.3, a inclusão (embedding) nos permite usar

um atalho sintático para definir um **ColoredPoint** que contém todos os campos de **Point**, além de outros. Se quisermos, podemos seleccionar os campos de **ColoredPoint** incluídos com **Point** sem mencionar essa estrutura:

```
var cp ColoredPoint
cp.X = 1
fmt.Println(cp.Point.X) // "1"
cp.Point.Y = 2
fmt.Println(cp.Y) // "2"
```

Um sistema semelhante aplica-se aos *métodos* de **Point**. Podemos chamar métodos do campo **Point** incluído usando um receptor do tipo **ColoredPoint**, apesar de **ColoredPoint** não ter métodos declarados:

```
red := color.RGBA{255, 0, 0, 255}
blue := color.RGBA{0, 0, 255, 255}
var p = ColoredPoint{Point{1, 1}, red}
var q = ColoredPoint{Point{5, 4}, blue}
fmt.Println(p.Distance(q.Point)) // "5"
p.ScaleBy(2)
q.ScaleBy(2)
fmt.Println(p.Distance(q.Point)) // "10"
```

Os métodos de **Point** foram *promovidos* para **ColoredPoint**. Desse modo, a inclusão permite que tipos complexos com muitos métodos sejam criados por *composição* de vários campos, cada um oferecendo alguns métodos.

Leitores familiarizados com linguagens orientadas a objetos baseadas em classes podem se sentir tentados a ver **Point** como uma classe-base e **ColoredPoint** como uma subclasse ou classe derivada, ou interpretar o relacionamento entre esses tipos como se um **ColoredPoint** “fosse” um **Point**. Porém, isso seria um erro. Observe as chamadas anteriores a **Distance**. **Distance** tem um parâmetro do tipo **Point**, e **q** não é um **Point**, portanto, embora **q** tenha um campo desse tipo incluído, devemos seleccioná-lo explicitamente. Tentar passar **q** seria um erro:

```
p.Distance(q) // erro de compilação: não é possível usar q (ColoredPoint)
               // como Point
```

Um **ColoredPoint** não é um **Point**, mas “tem” um **Point**, e tem dois

métodos adicionais, **Distance** e **ScaleBy**, promovidos a partir de **Point**. Se preferirmos pensar em termos de implementação, o campo incluído instrui o compilador a gerar métodos wrapper adicionais que delegam aos métodos declarados, o que é equivalente a:

```
func (p ColoredPoint) Distance(q Point) float64 {
    return p.Point.Distance(q)
}

func (p *ColoredPoint) ScaleBy(factor float64) {
    p.Point.ScaleBy(factor)
}
```

Quando **Point.Distance** é chamado pelo primeiro desses métodos wrapper, o valor de seu receptor não é **p**, mas **p.Point**, e não há como o método acessar o **ColoredPoint** em que **Point** está incluído.

O tipo de um campo anônimo pode ser um *ponteiro* para um tipo nomeado, caso em que os campos e os métodos são promovidos indiretamente a partir do objeto apontado. Acrescentar outro nível de acesso indireto permite compartilhar estruturas comuns e variar os relacionamentos entre os objetos dinamicamente. A declaração a seguir de **ColoredPoint** inclui um ***Point**:

```
type ColoredPoint struct {
    *Point
    Color color.RGBA
}

p := ColoredPoint{&Point{1, 1}, red}
q := ColoredPoint{&Point{5, 4}, blue}
fmt.Println(p.Distance(*q.Point)) // "5"
q.Point = p.Point                // p e q agora compartilham o mesmo Point
p.ScaleBy(2)
fmt.Println(*p.Point, *q.Point)  // "{2 2} {2 2}"
```

Um tipo estrutura pode ter mais de um campo anônimo. Se tivéssemos declarado **ColoredPoint** como:

```
type ColoredPoint struct {
    Point
    color.RGBA
}
```

um valor desse tipo teria todos os métodos de **Point**, todos os métodos de **RGBA** e qualquer método adicional declarado diretamente em **ColoredPoint**. Quando o compilador resolve um seletor como **p.ScaleBy** como um método, ele procura de início um método diretamente declarado chamado **ScaleBy**; em seguida procura métodos promovidos uma vez a partir dos campos incluídos de **ColoredPoint** e, então, métodos promovidos duas vezes a partir de campos incluídos em **Point** e em **RGBA**, e assim sucessivamente. O compilador informa um erro se o seletor for ambíguo porque dois métodos foram promovidos do mesmo nível.

Métodos podem ser declarados somente em tipos nomeados (como **Point**) e ponteiros para eles (***Point**), mas, graças à inclusão, é possível – e, às vezes, conveniente – que tipos estrutura *não nomeados* também tenham métodos.

A seguir, apresentamos um truque interessante para ilustrar a questão. Este exemplo mostra parte de um cache simples implementado com duas variáveis de nível de pacote, um mutex (seção 9.2) e o mapa que ele armazena:

```
var (
    mu sync.Mutex // guarda o mapeamento
    mapping = make(map[string]string)
)

func Lookup(key string) string {
    mu.Lock()
    v := mapping[key]
    mu.Unlock()
    return v
}
```

A versão a seguir é funcionalmente equivalente, porém agrupa as duas variáveis relacionadas em uma única variável de nível de pacote, **cache**:

```
var cache = struct {
    sync.Mutex
    mapping map[string]string
} {
    mapping: make(map[string]string),
}
```

```

}
func Lookup(key string) string {
    cache.Lock()
    v := cache.mapping[key]
    cache.Unlock()
    return v
}

```

A nova variável dá nomes mais expressivos às variáveis relacionadas ao `cache` e, como o campo `sync.Mutex` está incluído nele, seus métodos `Lock` e `Unlock` são promovidos para o tipo estrutura sem nome, permitindo travar `cache` com uma sintaxe autoexplicativa.

6.4 Valores e expressões método

Em geral, selecionamos e chamamos um método na mesma expressão, como em `p.Distance()`, mas é possível separar essas duas operações. O seletor `p.Distance` produz um *valor método* (*method value*): uma função que vincula um método (`Point.Distance`) a um valor específico de receptor `p`. Essa função pode então ser chamada sem um valor de receptor; ela só precisa dos argumentos que não são o receptor.

```

p := Point{1, 2}
q := Point{4, 6}

distanceFromP := p.Distance      // valor do método
fmt.Println(distanceFromP(q))    // "5"
var origin Point                 // {0, 0}
fmt.Println(distanceFromP(origin)) // "2.23606797749979", ;5

scaleP := p.ScaleBy // valor do método
scaleP(2)           // p torna-se (2, 4)
scaleP(3)           // em seguida (6, 12)
scaleP(10)          // em seguida (60, 120)

```

Valores método são úteis quando a API de um pacote precisa de um valor função e o comportamento desejado do cliente para essa função é chamar um método em um receptor específico. Por exemplo, a função `time.AfterFunc` chama um valor função após um intervalo de tempo especificado. O programa a seguir a usa para lançar o foguete `r` após dez segundos:

```

type Rocket struct { /* ... */ }
func (r *Rocket) Launch() { /* ... */ }

r := new(Rocket)
time.AfterFunc(10 * time.Second, func() { r.Launch() })

```

A sintaxe com o valor do método é mais concisa:

```
time.AfterFunc(10 * time.Second, r.Launch)
```

Relacionado ao valor do método, temos a *expressão método* (*method expression*). Ao chamar um método, em oposição a uma função comum, devemos fornecer o receptor de forma especial usando a sintaxe de seletor. Uma expressão método, escrita como `T.f` ou `(*T).f`, em que `T` é um tipo, produz um valor função com um primeiro parâmetro normal no lugar do receptor, portanto pode ser chamada da forma usual.

```

p := Point{1, 2}
q := Point{4, 6}

distance := Point.Distance // expressão método
fmt.Println(distance(p, q)) // "5"
fmt.Printf("%T\n", distance) // "func(Point, Point) float64"

scale := (*Point).ScaleBy
scale(&p, 2)
fmt.Println(p) // "{2 4}"
fmt.Printf("%T\n", scale) // "func(*Point, float64)"

```

Expressões método podem ser úteis quando precisamos de um valor para representar uma opção entre vários métodos pertencentes ao mesmo tipo, de modo que se possa chamar o método escolhido com vários receptores diferentes. No exemplo a seguir, a variável `op` representa o método de adição ou de subtração do tipo `Point`, e `Path.TranslateBy` o chama para cada ponto de `Path`:

```

type Point struct{ X, Y float64 }

func (p Point) Add(q Point) Point { return Point{p.X + q.X, p.Y + q.Y} }
func (p Point) Sub(q Point) Point { return Point{p.X - q.X, p.Y - q.Y} }

type Path []Point

func (path Path) TranslateBy(offset Point, add bool) {
    var op func(p, q Point) Point
    if add {
        op = Point.Add
    }
}

```

```

    } else {
        op = Point.Sub
    }
    for i := range path {
        // Chama path[i].Add(offset) ou path[i].Sub(offset).
        path[i] = op(path[i], offset)
    }
}

```

6.5 Exemplo: tipo vetor de bits

Conjuntos em Go normalmente são implementados como um `map[T]bool`, em que `T` é o tipo do elemento. Um conjunto representado por um mapa é bem flexível, mas, para determinados problemas, uma representação especializada pode ser melhor. Por exemplo, um *vetor de bits* (bit vector) é ideal em domínios como análise de fluxo de dados, em que os elementos do conjunto são inteiros pequenos não negativos, os conjuntos têm muitos elementos e as operações de conjuntos como união e intersecção são comuns.

Um vetor de bits usa uma fatia de valores inteiros sem sinal ou “palavras” (words), em que cada bit representa um possível elemento do conjunto. O conjunto contém *i* se o *i*-ésimo bit estiver ligado. O programa a seguir mostra um tipo simples de vetor de bits com três métodos:

gopl.io/ch6/intset

```

// Um IntSet é um conjunto de inteiros não negativos pequenos.
// Seu valor zero representa o conjunto vazio.
type IntSet struct {
    words []uint64
}

// Has informa se o conjunto contém o valor não negativo x.
func (s *IntSet) Has(x int) bool {
    word, bit := x/64, uint(x%64)
    return word < len(s.words) && s.words[word]&(1<<bit) != 0
}

// Add adiciona o valor não negativo x ao conjunto.
func (s *IntSet) Add(x int) {
    word, bit := x/64, uint(x%64)

```



```

    for word >= len(s.words) {
        s.words = append(s.words, 0)
    }
    s.words[word] |= 1 << bit
}

// UnionWith define s como a união de s e t.
func (s *IntSet) UnionWith(t *IntSet) {
    for i, tword := range t.words {
        if i < len(s.words) {
            s.words[i] |= tword
        } else {
            s.words = append(s.words, tword)
        }
    }
}

```

Como cada palavra tem 64 bits, para localizar o bit para x usamos o quociente $x/64$ como o índice da palavra e o resto $x\%64$ como o índice do bit dentro dela. A operação **UnionWith** utiliza o operador OU bit a bit `|` para calcular a união de 64 elementos a cada vez. (Retomaremos a escolha de palavras de 64 bits no exercício 6.5.)

Nessa implementação faltam vários recursos desejáveis, alguns dos quais são propostos como exercícios a seguir, mas um deles é essencial: uma forma de exibir um **IntSet** como uma string. Vamos implementar um método **String**, como fizemos com **Celsius** na seção 2.5:

```

// String devolve o conjunto como uma string no formato "{1 2 3}".
func (s *IntSet) String() string {
    var buf bytes.Buffer
    buf.WriteByte('{')
    for i, word := range s.words {
        if word == 0 {
            continue
        }
        for j := 0; j < 64; j++ {
            if word&(1<<uint(j)) != 0 {
                if buf.Len() > len("{}") {
                    buf.WriteByte(' ')
                }
                fmt.Fprintf(&buf, "%d", 64*i+j)
            }
        }
    }
}

```

```

    }
}
buf.WriteByte('}')
return buf.String()
}

```

Observe a semelhança entre o método **String** anterior e **intsToString** da seção 3.5.4; **bytes.Buffer** é frequentemente usado dessa maneira em métodos **String**. O pacote **fmt** trata tipos com um método **String** de forma especial para que valores de tipos complexos possam ser exibidos de forma amigável ao usuário. Em vez de exibir a representação bruta do valor (uma estrutura, nesse caso), **fmt** chama o método **String**. O sistema conta com interfaces e asserções de tipo, que explicaremos no capítulo 7.

Agora podemos mostrar **IntSet** em ação:

```

var x, y IntSet
x.Add(1)
x.Add(144)
x.Add(9)
fmt.Println(x.String()) // "{1 9 144}"

y.Add(9)
y.Add(42)
fmt.Println(y.String()) // "{9 42}"

x.UnionWith(&y)
fmt.Println(x.String()) // "{1 9 42 144}"

fmt.Println(x.Has(9), x.Has(123)) // "true false"

```

Uma advertência: declaramos **String** e **Has** como métodos do tipo ponteiro ***IntSet** não por necessidade, mas por questões de consistência em relação aos outros dois métodos que precisam de um receptor do tipo ponteiro, pois fazem atribuição a **s.words**. Consequentemente, um *valor* **IntSet** não tem um método **String**, o que, às vezes, pode resultar em surpresas como esta:

```

fmt.Println(&x)           // "{1 9 42 144}"
fmt.Println(x.String())   // "{1 9 42 144}"
fmt.Println(x)            // "[4398046511618 0 65536]"

```

No primeiro caso, exibimos um ponteiro ***IntSet**, que tem um método

String. No segundo, chamamos **String()** em uma variável **IntSet**; o compilador insere a operação **&** implícita, dando-nos um ponteiro, que tem o método **String**. Porém, no terceiro caso, como o valor **IntSet** não tem um método **String**, **fmt.Println** exibe a representação da estrutura. É importante não se esquecer do operador **&**. Fazer de **String** um método de **IntSet** e não de ***IntSet** pode ser uma boa ideia, mas é uma situação a ser avaliada caso a caso.

Exercício 6.1: Implemente os métodos adicionais a seguir:

```
func (*IntSet) Len() int      // devolve o número de elementos
func (*IntSet) Remove(x int) // remove x do conjunto
func (*IntSet) Clear()       // remove todos os elementos do conjunto
func (*IntSet) Copy() *IntSet // devolve uma cópia do conjunto
```

Exercício 6.2: Defina um método variádico **(*IntSet).AddAll(...int)** que aceite múltiplos valores a serem incluídos no conjunto, por exemplo, **s.AddAll(1, 2, 3)**.

Exercício 6.3: **(*IntSet).UnionWith** calcula a união de dois conjuntos usando **|**, o operador bit a bit OU para palavras paralelas. Implemente métodos para **IntersectWith**, **DifferenceWith** e **SymmetricDifference** para as operações correspondentes de conjunto. (A diferença simétrica entre dois conjuntos contém os elementos presentes em um conjunto ou no outro, mas não em ambos.)

Exercício 6.4: Acrescente um método **Elms** que devolva uma fatia contendo os elementos do conjunto, adequados para uma iteração com um loop **range**.

Exercício 6.5: O tipo de cada palavra usada por **IntSet** é **uint64**, mas a aritmética com 64 bits pode ser ineficiente em uma plataforma de 32 bits. Modifique o programa para que use o tipo **uint**, que é o tipo inteiro sem sinal mais eficiente para a plataforma. Em vez de dividir por 64, defina uma constante que armazene o tamanho real de **uint** em bits, isto é, 32 ou 64. Talvez você possa usar a expressão excessivamente esperta **32 << (^uint(0) >> 63)** para isso.

6.6 Encapsulamento

Dizemos que uma variável ou método de um objeto está *encapsulado* se for inacessível aos clientes do objeto. O encapsulamento, às vezes chamado de *ocultação de informações* (information hiding), é um aspecto fundamental da programação orientada a objetos.

Go tem apenas um mecanismo para controlar a visibilidade dos nomes: identificadores com letras maiúsculas são exportados do pacote em que estão definidos, enquanto nomes com letras minúsculas não são. O mesmo sistema que limita o acesso aos membros de um pacote também limita o acesso aos campos de uma estrutura ou aos métodos de um tipo. Consequentemente, para encapsular um objeto, devemos transformá-lo em uma estrutura.

Esse é o motivo pelo qual o tipo **IntSet** da seção anterior foi declarado como uma estrutura, apesar de ter apenas um único campo:

```
type IntSet struct {  
    words []uint64  
}
```

De modo alternativo, poderíamos ter definido **IntSet** como um tipo fatia, como mostrado a seguir, embora, é claro, devêssemos substituir toda ocorrência de **s.words** por ***s** em seus métodos:

```
type IntSet []uint64
```

Embora essa versão de **IntSet** seja essencialmente equivalente, ela permitiria que clientes de outros pacotes lessem e modificassem diretamente a fatia. Em outras palavras, enquanto a expressão ***s** poderia ser usada em qualquer pacote, **s.words** apareceria somente no pacote em que **IntSet** estivesse definido.

Outra consequência desse sistema baseado em nomes é que a unidade de encapsulamento é o pacote, e não o tipo, como em muitas outras linguagens. Os campos de uma estrutura são visíveis a todo código que estiver no mesmo pacote. O fato de o código estar em uma função ou em método não faz diferença.

O encapsulamento oferece três vantagens. Em primeiro lugar, como os

clientes não podem modificar diretamente as variáveis do objeto, é necessário inspecionar menos instruções para entender os possíveis valores dessas variáveis.

Em segundo, ocultar detalhes de implementação evita que os clientes dependam de aspectos que possam mudar, o que dá ao designer mais liberdade para evoluir com a implementação sem quebrar a compatibilidade com a API.

Como exemplo, considere o tipo **bytes.Buffer**, frequentemente usado para juntar strings bem curtas. Uma otimização que vale a pena fazer é reservar um pouco de espaço extra no objeto a fim de evitar alocação de memória nesse caso comum. Como **Buffer** é um tipo estrutura, esse espaço assume a forma de um campo extra do tipo **[64]byte** com um nome com inicial minúscula. Quando esse campo é acrescentado, pelo fato de não ser exportado, os clientes de **Buffer** fora do pacote **bytes** não percebem nenhuma mudança, exceto a melhoria no desempenho. **Buffer** e seu método **Grow** são apresentados a seguir, simplificados por questões de clareza:

```
type Buffer struct {
    buf    []byte
    initial [64]byte
    /* ... */
}

// Grow expande a capacidade do buffer, se for necessário,
// para garantir espaço para outros n bytes. [...]
func (b *Buffer) Grow(n int) {
    if b.buf == nil {
        b.buf = b.initial[:0] // usa o espaço pré-alocado inicialmente
    }
    if len(b.buf)+n > cap(b.buf) {
        buf := make([]byte, b.Len(), 2*cap(b.buf) + n)
        copy(buf, b.buf)
        b.buf = buf
    }
}
```

A terceira vantagem do encapsulamento e, em muitos casos, a mais importante, é evitar que clientes alterem arbitrariamente as variáveis de

um objeto. Como as variáveis do objeto podem ser modificadas somente por funções no mesmo pacote, o autor deste pode garantir que tais funções preservarão as invariantes internas do objeto. Por exemplo, o tipo **Counter** a seguir permite que os clientes incrementem o contador ou o reiniciem com zero, mas não permitem que ele receba um valor arbitrário:

```
type Counter struct { n int }  
  
func (c *Counter) N() int    { return c.n }  
func (c *Counter) Increment() { c.n++ }  
func (c *Counter) Reset()    { c.n = 0 }
```

Funções que simplesmente acessam ou modificam valores internos de um tipo, como os métodos do tipo **Logger** do pacote **log** a seguir, são chamados de *getters* e *setters*. No entanto, ao nomear um método getter, em geral omitimos o prefixo **Get**. Essa preferência por concisão se estende a todos os métodos, e não só aos métodos de acesso a campos, além de valer para outros prefixos redundantes também, como **Fetch**, **Find** e **Lookup**.

```
package log  
  
type Logger struct {  
    flags int  
    prefix string  
    // ...  
}  
  
func (l *Logger) Flags() int  
func (l *Logger) SetFlags(flag int)  
func (l *Logger) Prefix() string  
func (l *Logger) SetPrefix(prefix string)
```

O estilo de Go não proíbe campos exportados. É claro que, uma vez exportado, um campo não pode deixar de sê-lo sem uma mudança incompatível na API, portanto a escolha inicial deve ser cuidadosa e considerar a complexidade das invariantes que devem ser mantidas, a probabilidade de mudanças futuras e a quantidade de código cliente que será afetado por uma mudança.

O encapsulamento nem sempre é desejável. Ao expor sua representação como um número de nanossegundos do tipo **int64**, **time.Duration**

permite usar todas as operações comuns de aritmética e de comparação com durações e até mesmo definir constantes do tipo:

```
const day = 24 * time.Hour
fmt.Println(day.Seconds()) // "86400"
```

Como outro exemplo, compare **IntSet** com o tipo **geometry.Path** do início deste capítulo. **Path** foi definido como um tipo fatia, permitindo que seus clientes criem instâncias usando a sintaxe de fatia literal para iterar por seus pontos usando um loop **range** e assim por diante, enquanto essas operações não são permitidas para clientes de **IntSet**.

Eis a diferença crucial: **geometry.Path** é intrinsecamente uma sequência de pontos, nada mais nada menos, e não prevemos a adição de novos campos a ele. Portanto, faz sentido que o pacote **geometry** revele que **Path** é uma fatia. Em comparação, um **IntSet** por acaso foi representado como uma fatia `[]uint64`. Ele poderia ter sido representado com `[]uint` ou com algo totalmente diferente para conjuntos que são esparsos ou bem pequenos, e talvez possa se beneficiar de recursos adicionais, como um campo extra para registrar o número de elementos do conjunto. Por esses motivos, faz sentido que **IntSet** seja opaco.

Neste capítulo, aprendemos a associar métodos a tipos nomeados e a chamar esse métodos. Embora os métodos sejam fundamentais na programação orientada a objetos, eles constituem apenas metade do quadro geral. Para completá-lo, precisamos das *interfaces*, que são o assunto do próximo capítulo.

7

Interfaces

Os tipos interface expressam generalizações ou abstrações sobre os comportamentos de outros tipos. Ao generalizar, as interfaces permitem escrever funções mais flexíveis e adaptáveis porque não estão amarradas aos detalhes de uma implementação em particular.

Muitas linguagens orientadas a objetos têm alguma noção de interfaces, mas o que torna as interfaces de Go tão distintas é que elas são *satisfeitas implicitamente*. Em outras palavras, não há necessidade de declarar todas as interfaces que um tipo concreto satisfaz; é suficiente apenas ter os métodos necessários. Esse design permite criar novas interfaces que são satisfeitas por tipos concretos existentes sem alterar esses tipos, o que é particularmente útil para tipos definidos em pacotes sobre os quais você não tem controle.

Neste capítulo, começaremos analisando o funcionamento básico dos tipos interface e seus valores. Estudaremos várias interfaces importantes da biblioteca-padrão. Muitos programas Go fazem uso tanto de interfaces-padrão quanto de interfaces próprias. Por fim, veremos *asserções de tipo* (seção 7.10) e *switches de tipo* (seção 7.13) e como eles permitem um tipo diferente de generalização.

7.1 Interfaces como contratos

Todos os tipos que vimos até agora eram *tipos concretos*. Um tipo concreto especifica a representação exata de seus valores e expõe as operações intrínsecas dessa representação, como a aritmética para números, ou indexação, **append** e **range** para fatias. Esse tipo também pode oferecer comportamentos adicionais por meio de seus métodos. Quando temos um valor para um tipo concreto, sabemos exatamente o que ele é e o que

podemos *fazer* com ele.

Há outra espécie de tipo em Go que se chama *tipo interface*. Uma interface é um *tipo abstrato*. Ela não expõe a representação ou a estrutura interna de seus valores nem o conjunto de operações básicas que aceita; apenas revela alguns de seus métodos. Quando tiver um valor de um tipo interface, você não saberá nada sobre o que ele *é*, mas somente o que ele pode *fazer* ou, mais precisamente, quais comportamentos são oferecidos por seus métodos.

Ao longo deste livro, usamos duas funções semelhantes para formatação de string: `fmt.Printf`, que escreve o resultado na saída-padrão (um arquivo), e `fmt.Sprintf`, que devolve o resultado como uma `string`. Seria lamentável se a parte difícil – a formatação do resultado – tivesse de ser duplicada por causa dessas diferenças superficiais sobre como o resultado é usado. Graças às interfaces, isso não ocorre. As duas funções, na verdade, embrulham uma terceira função, `fmt.Fprintf`, que não toma conhecimento do que acontece com o resultado que ela calcula:

```
package fmt

func Fprintf(w io.Writer, format string, args ...interface{}) (int, error)

func Printf(format string, args ...interface{}) (int, error) {
    return Fprintf(os.Stdout, format, args...)
}

func Sprintf(format string, args ...interface{}) string {
    var buf bytes.Buffer
    Fprintf(&buf, format, args...)
    return buf.String()
}
```

O prefixo F de `Fprintf` quer dizer *file* (arquivo) e indica que a saída formatada deve ser escrita no arquivo fornecido como primeiro argumento. No caso de `Printf`, o argumento `os.Stdout` é um `*os.File`. No caso de `Sprintf`, porém, o argumento não é um arquivo, embora, superficialmente, lembre um: `&buf` é um ponteiro para um buffer de memória no qual os bytes podem ser escritos.

O primeiro parâmetro de `Fprintf` também não é um arquivo. É um

`io.Writer`, que é um tipo interface com a seguinte declaração:

```
package io

// Writer é a interface que inclui o método básico Write.
type Writer interface {
    // Write escreve len(p) bytes de p no stream de dados subjacente.
    // Ele devolve o número de bytes escritos de p (0 <= n <= len(p))
    // e qualquer erro encontrado que faça a escrita ser interrompida.
    // Write deve retornar um erro diferente de nil se devolver n < len(p).
    // Write não deve modificar os dados da fatia, mesmo que temporariamente.
    //
    // As implementações não devem reter p.
    Write(p []byte) (n int, err error)
}
```

A interface `io.Writer` define o contrato entre `Fprintf` e quem a chama. Por um lado, o contrato exige que quem chama forneça um valor de um tipo concreto como `*os.File` ou `*bytes.Buffer`, que tenha um método chamado `Write` com a assinatura e o comportamento apropriados. Por outro, o contrato garante que `Fprintf` fará seu trabalho, dado qualquer valor que satisfaça a interface `io.Writer`. `Fprintf` não deve supor que está escrevendo em um arquivo ou na memória, mas apenas que pode chamar `Write`.

Como `fmt.Fprintf` não supõe nada sobre a representação do valor e conta somente com os comportamentos garantidos pelo contrato de `io.Writer`, podemos com segurança passar um valor de qualquer tipo concreto que satisfaça `io.Writer` como primeiro argumento de `fmt.Fprintf`. Essa liberdade de substituir um tipo por outro que satisfaça a mesma interface chama-se *substituibilidade* (substitutability) e é uma marca registrada da programação orientada a objetos.

Vamos testar isso usando um tipo novo. O método `Write` do tipo `*ByteCounter` a seguir simplesmente conta os bytes escritos antes de descartá-los. (A conversão é necessária para fazer os tipos de `len(p)` e de `*c` serem iguais na instrução de atribuição `+=`.)

gopl.io/ch7/bytecounter

```
type ByteCounter int
```

```
func (c *ByteCounter) Write(p []byte) (int, error) {
    *c += ByteCounter(len(p)) // converte int para ByteCounter
    return len(p), nil
}
```

Como ***ByteCounter** satisfaz o contrato de **io.Writer**, podemos passá-lo para **Fprintf**, que faz sua formatação de string sem tomar conhecimento dessa mudança; **ByteCounter** acumula corretamente o tamanho do resultado.

```
var c ByteCounter
c.Write([]byte("hello"))
fmt.Println(c) // "5", = len("hello")

c = 0 // reinicia o contador
var name = "Dolly"
fmt.Fprintf(&c, "hello, %s", name)
fmt.Println(c) // "12", = len("hello, Dolly")
```

Além de **io.Writer**, há outra interface muito importante para o pacote **fmt**. **Fprintf** e **Fprintln** oferecem uma maneira de os tipos controlarem o modo como seus valores são exibidos. Na seção 2.5, definimos um método **String** para o tipo **Celsius** de modo que as temperaturas fossem exibidas como **"100°C"**; na seção 6.5, equipamos ***IntSet** com um método **String** para que os conjuntos pudessem ser apresentados com a notação tradicional de conjuntos, como **"{1 2 3}"**. Declarar um método **String** faz um tipo satisfazer **fmt.Stringer**, que é uma das interfaces mais amplamente usadas:

```
package fmt

// O método String é usado para exibir valores passados
// como um operando para qualquer formato que aceite uma string
// ou para uma função de saída não formatada, como Print.
type Stringer interface {
    String() string
}
```

Explicaremos como o pacote **fmt** descobre quais valores satisfazem essa interface na seção 7.10.

Exercício 7.1: Usando as ideias de **ByteCounter**, implemente contadores para palavras e linhas. Você perceberá que **bufio.ScanWords** pode ser

útil.

Exercício 7.2: Escreva uma função `CountingWriter` com a assinatura a seguir que, dado um `io.Writer`, devolva um novo `Writer` que encapsule o original, e um ponteiro para uma variável `int64` que, em qualquer instante, contenha o número de bytes escritos no novo `Writer`.

```
func CountingWriter(w io.Writer) (io.Writer, *int64)
```

Exercício 7.3: Escreva um método `String` para o tipo `*tree` em `gopl.io/ch4/treesort` (seção 4.4) que revele a sequência de valores da árvore.

7.2 Tipos interface

Um tipo interface especifica um conjunto de métodos que um tipo concreto deve ter para ser considerado uma instância dessa interface.

O tipo `io.Writer` é uma das interfaces mais utilizadas, pois ela oferece uma abstração para todos os tipos no qual bytes podem ser escritos, o que inclui arquivos, buffers em memória, conexões de rede, clientes HTTP, compressores de arquivos, hashers, e assim por diante. O pacote `io` define várias outras interfaces úteis. Um `Reader` representa qualquer tipo a partir do qual você pode ler bytes, e um `Closer` é qualquer valor que pode ser fechado, por exemplo, um arquivo ou uma conexão de rede. (A essa altura, provavelmente você já deve ter notado a convenção de nomenclatura das interfaces de método único em Go.)

```
package io

type Reader interface {
    Read(p []byte) (n int, err error)
}

type Closer interface {
    Close() error
}
```

Se observarmos mais além, encontramos declarações de novos tipos interface como combinações de tipos existentes. Aqui estão dois exemplos:

```

type ReadWriter interface {
    Reader
    Writer
}

type ReadWriteCloser interface {
    Reader
    Writer
    Closer
}

```

A sintaxe usada anteriormente, que lembra a inclusão de estruturas (struct embedding), permite nomear outra interface concisamente para escrever todos os seus métodos. Isso é chamado de *inclusão* (embedding) de interface. Poderíamos ter escrito `io.ReadWriter` sem inclusão de interfaces, embora ficasse menos sucinto, desta maneira:

```

type ReadWriter interface {
    Read(p []byte) (n int, err error)
    Write(p []byte) (n int, err error)
}

```

ou até mesmo usando uma combinação dos dois estilos:

```

type ReadWriter interface {
    Read(p []byte) (n int, err error)
    Writer
}

```

Todas as três declarações têm o mesmo efeito. A ordem em que os métodos aparecem não importa. Tudo que interessa é o conjunto de métodos.

Exercício 7.4: A função `strings.NewReader` devolve um valor que satisfaz a interface `io.Reader` (e outras) lendo de seu argumento, que é uma string. Implemente uma versão simples de `NewReader` e use-a para fazer o parser HTML (seção 5.2) aceitar entrada de uma string.

Exercício 7.5: A função `LimitReader` do pacote `io` aceita um `io.Reader` `r` e um número de bytes `n` e devolve outro `Reader` que lê de `r`, mas informa uma condição de fim de arquivo (end-of-file) após `n` bytes. Implemente-a.

```

func LimitReader(r io.Reader, n int64) io.Reader

```

7.3 Como satisfazer uma interface

Um tipo *satisfaz* uma interface se possuir todos os métodos exigidos por ela. Por exemplo, um `*os.File` satisfaz `io.Reader`, `Writer`, `Closer` e `ReadWrite`. Um `*bytes.Buffer` satisfaz `Reader`, `Writer` e `ReadWrite`, mas não satisfaz `Closer` porque não tem um método `Close`. Por concisão, programadores Go muitas vezes dizem que um tipo concreto “é” um tipo particular de interface, o que quer dizer que ele satisfaz a interface. Por exemplo, um `*bytes.Buffer` é um `io.Writer`; um `*os.File` é um `io.ReadWriter`.

A regra de possibilidade de atribuição (seção 2.4.2) para interfaces é bem simples: uma expressão pode ser atribuída a uma interface somente se seu tipo satisfizer a interface. Portanto:

```
var w io.Writer
w = os.Stdout           // OK: *os.File tem um método Write
w = new(bytes.Buffer)   // OK: *bytes.Buffer tem um método Write
w = time.Second         // erro de compilação: time.Duration não tem um método Write

var rwc io.ReadWriteCloser
rwc = os.Stdout         // OK: *os.File tem métodos Read, Write e Close
rwc = new(bytes.Buffer) // erro de compilação: *bytes.Buffer não tem um
                        // método Close
```

Essa regra se aplica até mesmo quando o próprio lado direito é uma interface:

```
w = rwc                // OK: io.ReadWriteCloser tem um método Write
rwc = w                // erro de compilação: io.Writer não tem um método Close
```

Como `ReadWrite` e `ReadWriteCloser` incluem todos os métodos de `Writer`, qualquer tipo que satisfaça `ReadWrite` ou `ReadWriteCloser` necessariamente satisfaz `Writer`.

Antes de prosseguir, devemos explicar uma sutileza a respeito do que significa ter um método para um tipo. De acordo com a seção 6.2, para cada tipo concreto chamado `T` alguns de seus métodos têm um receptor do próprio tipo `T`, enquanto outros exigem um ponteiro `*T`. Lembre-se de que é permitido chamar um método `*T` em um argumento de tipo `T`, desde que o argumento seja uma *variável*; o compilador implicitamente

toma seu endereço. Porém isso é apenas açúcar sintático: um valor de tipo `T` não possui todos os métodos que um ponteiro `*T` tem e, como resultado, pode satisfazer a menos interfaces.

Um exemplo esclarecerá isso. O método `String` do tipo `IntSet` da seção 6.5 exige um receptor do tipo ponteiro, portanto não podemos chamar esse método em um valor de `IntSet` não endereçável:

```
type IntSet struct { /* ... */ }
func (*IntSet) String() string

var _ = IntSet{}.String() // erro de compilação: String exige um
                          // receptor *IntSet
```

mas podemos chamá-lo em uma variável do tipo `IntSet`:

```
var s IntSet
var _ = s.String() // OK: s é uma variável e &s tem um método String
```

Todavia, como apenas `*IntSet` tem um método `String`, somente `*IntSet` satisfaz a interface `fmt.Stringer`:

```
var _ fmt.Stringer = &s // OK
var _ fmt.Stringer = s  // erro de compilação: IntSet não tem um método String
```

A seção 12.8 inclui um programa que exibe os métodos de um valor qualquer, e a ferramenta `godoc -analysis=type` (seção 10.7.4) apresenta os métodos de cada tipo e o relacionamento entre interfaces e tipos concretos.

Como um envelope que envolve e esconde a carta que ele armazena, uma interface envolve e oculta o tipo concreto e o valor que ele armazena. Somente os métodos revelados pelo tipo interface podem ser chamados, mesmo que o tipo concreto tenha outros métodos:

```
os.Stdout.Write([]byte("hello")) // OK: *os.File tem um método Write
os.Stdout.Close()                 // OK: *os.File tem um método Close

var w io.Writer
w = os.Stdout
w.Write([]byte("hello")) // OK: io.Writer tem um método Write
w.Close()                // erro de compilação: io.Writer não tem um
                          // método Close
```

Uma interface com mais métodos, como `io.ReadWriter`, diz mais sobre

os valores que ela contém e impõe mais exigências aos tipos que a implementam do que uma interface com menos métodos, como `io.Reader`. Então, o que o tipo `interface{}`, que não tem nenhum método, diz sobre os tipos concretos que o satisfazem?

É isso mesmo: nada. Isso pode parecer inútil, mas na verdade o tipo `interface{}`, que é conhecido como *interface vazia* (empty interface), é indispensável. Como a interface vazia não impõe nenhuma exigência sobre os tipos que a satisfazem, podemos atribuir *qualquer* valor a ela.

```
var any interface{}
any = true
any = 12.34
any = "hello"
any = map[string]int{"one": 1}
any = new(bytes.Buffer)
```

Embora não seja óbvio, usamos a interface vazia desde o primeiro exemplo deste livro, pois é isso que permite que funções como `fmt.Println` ou `errorf` na seção 5.7 aceitem argumentos de qualquer tipo.

É claro que, ao criar um valor `interface{}` contendo um booleano, um número de ponto flutuante, uma string, um mapa, um ponteiro ou qualquer outro tipo, não podemos fazer nada diretamente com o valor que ele armazena, pois a interface não tem nenhum método. Precisamos de uma maneira de obter outra vez o valor. Veremos como fazer isso usando uma *asserção de tipo* (type assertion) na seção 7.10.

Como satisfazer uma interface depende somente dos métodos dos dois tipos envolvidos, não há necessidade de declarar o relacionamento entre um tipo concreto e as interfaces que ele satisfaz. Apesar do que foi dito, ocasionalmente é conveniente documentar e fazer a asserção do relacionamento quando há a intenção de criar esse relacionamento, mas ele não é imposto pelo programa. A declaração a seguir garante, em tempo de compilação, que um valor de tipo `*bytes.Buffer` satisfaça `io.Writer`:

```
// *bytes.Buffer deve satisfazer io.Writer
```



```
var w io.Writer = new(bytes.Buffer)
```

Não precisamos alocar uma nova variável, pois qualquer valor de tipo `*bytes.Buffer` servirá, até mesmo `nil`, que escrevemos como `(*bytes.Buffer)(nil)`, usando uma conversão explícita. Como não pretendemos referenciar `w`, podemos substituí-lo pelo identificador vazio. Juntas, essas alterações nos dão a seguinte variante, mais econômica:

```
// *bytes.Buffer deve satisfazer io.Writer
var _ io.Writer = (*bytes.Buffer)(nil)
```

Tipos não vazios de interface, como `io.Writer`, são satisfeitos com mais frequência por um tipo ponteiro, em particular quando um ou mais métodos da interface implicam algum tipo de mutação no receptor, como faz o método `Write`. Um ponteiro para uma estrutura é um tipo especialmente comum que contém métodos associados.

Porém, tipos ponteiro de forma alguma são os únicos que satisfazem interfaces, e até mesmo interfaces com métodos que fazem mudanças (mutator) podem ser satisfeitas por um dos outros tipos de referência de Go. Vimos exemplos de tipos fatia com métodos (`geometry.Path`, seção 6.1) e tipos mapa com métodos (`url.Values` na seção 6.2.1) e, adiante, veremos um tipo função com métodos (`http.HandlerFunc` na seção 7.7). Até mesmo tipos básicos podem satisfazer interfaces; como veremos na seção 7.4, `time.Duration` satisfaz `fmt.Stringer`.

Um tipo concreto pode satisfazer muitas interfaces não relacionadas. Considere um programa que organiza ou vende artefatos culturais digitalizados como música, filmes e livros. Esse programa pode definir o seguinte conjunto de tipos concretos:

```
Album
Book
Movie
Magazine
Podcast
TVEpisode
Track
```

Podemos expressar cada abstração em que estamos interessados como uma interface. Algumas propriedades são comuns a todos os artefatos,

por exemplo, título, data de criação e uma lista de criadores (autores ou artistas).

```
type Artifact interface {
    Title() string
    Creators() []string
    Created() time.Time
}
```

Outras propriedades são restritas a determinados tipos de artefato. As propriedades da palavra impressa são relevantes apenas a livros e revistas, enquanto somente filmes e episódios de TV têm uma resolução de tela.

```
type Text interface {
    Pages() int
    Words() int
    PageSize() int
}
type Audio interface {
    Stream() (io.ReadCloser, error)
    RunningTime() time.Duration
    Format() string // por exemplo, "MP3", "WAV"
}
type Video interface {
    Stream() (io.ReadCloser, error)
    RunningTime() time.Duration
    Format() string // por exemplo, "MP4", "WMV"
    Resolution() (x, y int)
}
```

Essas interfaces nada mais são que uma maneira útil de agrupar tipos concretos relacionados e expressar os aspectos que eles têm em comum. Podemos descobrir outros agrupamentos posteriormente. Por exemplo, se percebermos a necessidade de lidar com itens de **Audio** e **Video** da mesma maneira, podemos definir uma interface **Streamer** para representar seus aspectos comuns sem alterar nenhuma declaração de tipo existente.

```
type Streamer interface {
    Stream() (io.ReadCloser, error)
    RunningTime() time.Duration
    Format() string
}
```

Cada agrupamento de tipos concretos baseado em seus comportamentos compartilhados pode ser expresso como um tipo interface. De modo diferente de linguagens baseadas em classe, em que o conjunto de interfaces satisfeito por uma classe é explícito, em Go podemos definir novas abstrações ou agrupamentos de interesse quando precisamos deles, sem modificar a declaração do tipo concreto. Isso é particularmente útil quando o tipo concreto é proveniente de um pacote escrito por um autor diferente. É claro, porém, que é preciso haver aspectos subjacentes comuns nos tipos concretos.

7.4 Fazendo parse de flags com `flag.Value`

Nesta seção, veremos como outra interface padrão, `flag.Value`, ajuda a definir novas notações para flags de linha de comando. Considere o programa a seguir, que dorme durante um período de tempo especificado.

gopl.io/ch7/sleep

```
var period = flag.Duration("period", 1*time.Second, "sleep period")

func main() {
    flag.Parse()
    fmt.Printf("Sleeping for %v...", *period)
    time.Sleep(*period)
    fmt.Println()
}
```

Antes de dormir, o programa exibe o período de tempo. O pacote `fmt` chama o método `String` de `time.Duration` para exibir o período, não com o número de nanossegundos, mas com uma notação mais amigável ao usuário:

```
$ go build gopl.io/ch7/sleep
$ ./sleep
Sleeping for 1s...
```

Por padrão, o período para dormir é de um segundo, mas pode ser controlado por meio da flag de linha de comando `-period`. A função `flag.Duration` cria uma variável flag do tipo `time.Duration` e permite que o usuário especifique a duração em vários formatos amigáveis ao

usuário, incluindo a mesma notação exibida pelo método **String**. Essa simetria de design resulta em uma boa interface de usuário.

```
$ ./sleep -period 50ms
Sleeping for 50ms...
$ ./sleep -period 2m30s
Sleeping for 2m30s...
$ ./sleep -period 1.5h
Sleeping for 1h30m0s...
$ ./sleep -period "1 day"
invalid value "1 day" for flag -period: time: invalid duration 1 day
```

Como flags de duração são muito úteis, esse recurso está incluído no pacote **flag**, mas é fácil definir novas notações de flags para nossos próprios tipos de dados. Basta definir um tipo que satisfaça a interface **flag.Value**, cuja declaração está apresentada a seguir:

```
package flag

// Value é a interface para o valor armazenado em uma flag.
type Value interface {
    String() string
    Set(string) error
}
```

O método **String** formata o valor da flag para usar em mensagens de ajuda da linha de comando; assim, todo **flag.Value** também é um **fmt.Stringer**. O método **Set** faz parse de seu argumento do tipo string e atualiza o valor da flag. Na verdade, o método **Set** é o inverso do método **String**, e usar a mesma notação para ambos é uma boa prática.

Vamos definir um tipo **celsiusFlag** que permite que uma temperatura seja especificada em Celsius, ou em Fahrenheit com uma conversão apropriada. Observe que **celsiusFlag** inclui um **Celsius** (seção 2.5), obtendo assim um método **String** gratuitamente. Para satisfazer **flag.Value**, precisamos apenas declarar o método **Set**:

gopl.io/ch7/tempconv

```
// *celsiusFlag satisfaz a interface flag.Value.
type celsiusFlag struct{ Celsius }

func (f *celsiusFlag) Set(s string) error {
    var unit string
```

```

var value float64
fmt.Sscanf(s, "%f%s", &value, &unit) // não é preciso fazer verificação
                                     // de erro

switch unit {
case "C", "°C":
    f.Celsius = Celsius(value)
    return nil
case "F", "°F":
    f.Celsius = FToC(Fahrenheit(value))
    return nil
}
return fmt.Errorf("invalid temperature %q", s)
}

```

A chamada a `fmt.Sscanf` faz parse de um número de ponto flutuante (`value`) e de uma string (`unit`) da entrada `s`. Embora geralmente seja necessário verificar o resultado de erro de `Sscanf`, nesse caso não é preciso fazer isso porque, se ocorrer um problema, não haverá correspondência com nenhum dos casos de `switch`.

A função `CelsiusFlag` a seguir engloba tudo. Para quem a chama, um ponteiro para o campo `Celsius` incluído da variável `f` do tipo `celsiusFlag` é devolvido. O campo `Celsius` é a variável que será atualizada pelo método `Set` durante o processamento das flags. A chamada a `Var` adiciona a flag ao conjunto de flags da linha de comando da aplicação, que é a variável global `flag.CommandLine`. Programas com interfaces de linha de comando singularmente complexas podem ter muitas variáveis desse tipo. A chamada a `Var` atribui um argumento `*celsiusFlag` a um parâmetro `flag.Value`, fazendo o compilador verificar se `*celsiusFlag` tem os métodos necessários.

```

// CelsiusFlag define uma flag Celsius com o nome especificado, o valor
// default e o uso, e devolve o endereço da variável de flag. O argumento
// flag deve ter uma quantidade e uma unidade, por exemplo, "100C".
func CelsiusFlag(name string, value Celsius, usage string) *Celsius {
    f := celsiusFlag{value}
    flag.CommandLine.Var(&f, name, usage)
    return &f.Celsius
}

```

Agora podemos começar a usar a nova flag em nossos programas:

gopl.io/ch7/tempflag

```
var temp = tempconv.CelsiusFlag("temp", 20.0, "the temperature")

func main() {
    flag.Parse()
    fmt.Println(*temp)
}
```

Eis uma sessão típica:

```
$ go build gopl.io/ch7/tempflag
$ ./tempflag
20°C
$ ./tempflag -temp -18C
-18°C
$ ./tempflag -temp 212°F
100°C
$ ./tempflag -temp 273.15K
invalid value "273.15K" for flag -temp: invalid temperature "273.15K"
Usage of ./tempflag:
    -temp value
        the temperature (default 20°C)
$ ./tempflag -help
Usage of ./tempflag:
    -temp value
        the temperature (default 20°C)
```

Exercício 7.6: Acrescente suporte para temperaturas em Kelvin a `tempflag`.

Exercício 7.7: Explique por que a mensagem de ajuda contém °C quando o valor default de 20.0 não tem.

7.5 Valores interface

Conceitualmente, um valor de um tipo interface, ou seja, um *valor interface*, tem dois componentes: um tipo concreto e um valor desse tipo. Eles são chamados de *tipo dinâmico* e *valor dinâmico* da interface.

Para uma linguagem estaticamente tipada como Go, tipos são um conceito de tempo de compilação, portanto um tipo não é um valor. Em nosso modelo conceitual, um conjunto de valores chamado *descritores de tipo* (type descriptors) oferece informações sobre cada tipo, por exemplo, seu nome e seus métodos. Em um valor interface, o componente do tipo é

representado pelo descritor de tipo apropriado.

Nas quatro instruções a seguir, a variável `w` assume três valores diferentes. (Os valores inicial e final são iguais.)

```
var w io.Writer
w = os.Stdout
w = new(bytes.Buffer)
w = nil
```

Vamos observar o valor e o comportamento dinâmico de `w` com mais detalhes após cada instrução. A primeira instrução declara `w`:

```
var w io.Writer
```

Em Go, variáveis são sempre inicializadas com um valor bem definido, e as interfaces não são exceção. O valor zero de uma interface tem os componentes para tipo e valor definidos com `nil` (Figura 7.1).

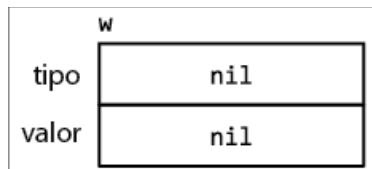


Figura 7.1 – Um valor interface nil.

Um valor interface é descrito como nil ou diferente de nil de acordo com seu tipo dinâmico, portanto esse é um valor interface nil. Você pode testar se um valor interface é nil usando `w == nil` ou `w != nil`. Chamar qualquer método de um valor interface nil gera um pânico:

```
w.Write([]byte("hello")) // pânico: desreferência de um ponteiro nil
```

A segunda instrução atribui um valor de tipo `*os.File` a `w`:

```
w = os.Stdout
```

Essa atribuição envolve uma conversão implícita de um tipo concreto para um tipo interface e é equivalente à conversão explícita `io.Writer(os.Stdout)`. Uma conversão desse tipo, seja explícita ou implícita, captura o tipo e o valor de seu operando. O tipo dinâmico do valor da interface é definido com o descritor de tipo para o tipo ponteiro `*os.File`, e seu valor dinâmico armazena uma cópia de `os.Stdout`, que é um ponteiro para a variável `os.File` representando a saída-padrão do

processo (Figura 7.2).

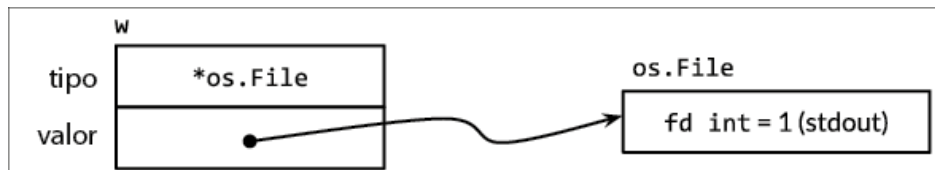


Figura 7.2 – Um valor interface contendo um ponteiro **os.File*.

Chamar o método **Write** em um valor interface contendo um ponteiro ***os.File** faz o método **(*os.File).Write** ser chamado. A chamada exibe "hello".

```
w.Write([]byte("hello")) // "hello"
```

Em geral, não podemos saber em tempo de compilação qual será o tipo dinâmico de um valor interface. Assim, uma chamada por meio de uma interface deve usar um *despacho dinâmico* (dynamic dispatch). Em vez de fazer uma chamada direta, o compilador deve gerar código para obter o endereço do método chamado **Write** a partir do descritor de tipo e então fazer uma chamada indireta a esse endereço. O argumento do receptor para a chamada é uma cópia do valor dinâmico da interface, isto é, **os.Stdout**. O efeito equivale a termos realizado essa chamada diretamente:

```
os.Stdout.Write([]byte("hello")) // "hello"
```

A terceira instrução atribui um valor do tipo ***bytes.Buffer** ao valor da interface:

```
w = new(bytes.Buffer)
```

O tipo dinâmico agora é ***bytes.Buffer**, e o valor dinâmico é um ponteiro para o recém-alocado buffer (Figura 7.3).

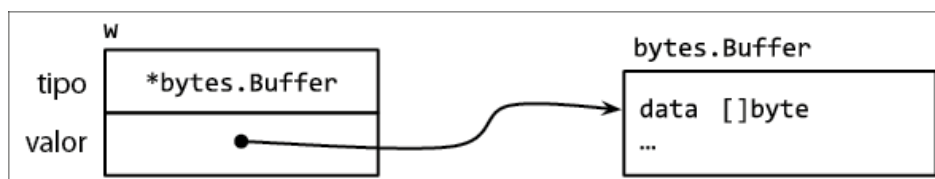


Figura 7.3 – Um valor interface contendo um ponteiro **bytes.Buffer*.

Uma chamada ao método **Write** usa o mesmo sistema anterior:


```
w.Write([]byte("hello")) // escreve "hello" em bytes.Buffer
```

Dessa vez, o descritor de tipo é `*bytes.Buffer`, portanto o método `(*bytes.Buffer).Write` é chamado, com o endereço do buffer como o valor do parâmetro do receptor. A chamada concatena "hello" ao buffer.

Por fim, a quarta instrução atribui `nil` ao valor da interface:

```
w = nil
```

Isso reinicia seus dois componentes com `nil`, restaurando `w` para o mesmo estado que tinha quando foi declarado, o que está mostrado na figura 7.1.

Um valor interface pode armazenar valores dinâmicos arbitrariamente grandes. Por exemplo, o tipo `time.Time`, que representa um instante no tempo, é um tipo estrutura com vários campos não exportados. Se criarmos um valor interface com ele:

```
var x interface{} = time.Now()
```

o resultado poderá ser semelhante ao que mostra a figura 7.4. Conceitualmente, o valor dinâmico sempre cabe no valor interface, independentemente do tamanho de seu tipo. (Esse é apenas um modelo conceitual; uma implementação realista é bem diferente.)

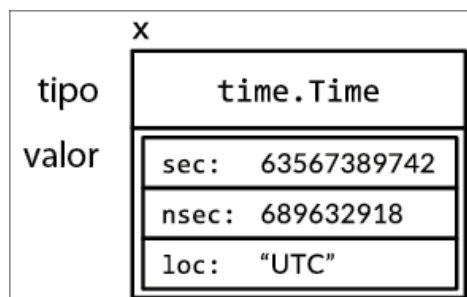


Figura 7.4 – Um valor interface armazenando uma estrutura time.Time.

Valores interface podem ser comparados com `==` e `!=`. Dois valores interface são iguais se ambos forem `nil` ou se seus tipos dinâmicos forem idênticos e seus valores dinâmicos forem iguais de acordo com o comportamento usual de `==` para esse tipo. Como valores interface são comparáveis, eles podem ser usados como chaves de um mapa ou como operando de uma instrução `switch`.

No entanto, se dois valores interface forem comparados e tiverem o mesmo tipo dinâmico, porém esse tipo não for comparável (uma fatia, por exemplo), a comparação falhará com um pânico:

```
var x interface{} = []int{1, 2, 3}
fmt.Println(x == x) // pânico: comparando o tipo []int que é incomparável
```

Nesse aspecto, os tipos interface são atípicos. Outros tipos são seguramente comparáveis (como os tipos básicos e ponteiros) ou não são comparáveis (como fatias, mapas e funções), mas quando comparamos valores interface ou tipos agregados que contenham valores interface devemos estar cientes do potencial para um pânico. Um risco semelhante existe quando usamos interfaces como chaves de mapas ou como operandos de switch. Somente compare valores interface se estiver certo de que eles contêm valores dinâmicos de tipos comparáveis.

Quando tratar erros ou durante a depuração, com frequência é útil informar o tipo dinâmico de um valor interface. Para isso, utilize o verbo `%T` do pacote `fmt`:

```
var w io.Writer
fmt.Printf("%T\n", w) // "<nil>"

w = os.Stdout
fmt.Printf("%T\n", w) // "*os.File"

w = new(bytes.Buffer)
fmt.Printf("%T\n", w) // "*bytes.Buffer"
```

Internamente, `fmt` usa reflexão (reflection) para obter o nome do tipo dinâmico da interface. Veremos a reflexão no capítulo 12.

7.5.1 Ressalva: uma interface contendo um ponteiro nil não é nil

Um valor interface nil que não contém valor não é o mesmo que um valor interface contendo um ponteiro que, por acaso, seja nil. Essa distinção sutil cria uma armadilha em que todo programador Go já caiu.

Considere o programa a seguir. Com `debug` definido com `true`, a função principal coleta a saída da função em um `bytes.Buffer`.

```
const debug = true
```

```

func main() {
    var buf *bytes.Buffer
    if debug {
        buf = new(bytes.Buffer) // habilita a coleta da saída
    }
    f(buf) // NOTA: sutilmente incorreto!
    if debug {
        // ...usa buf...
    }
}

// Se out for diferente de nil, a saída será escrita nela.
func f(out io.Writer) {
    // ...faz algo...
    if out != nil {
        out.Write([]byte("done!\n"))
    }
}

```

Poderíamos esperar que alterar **debug** para **false** desabilitaria a coleta da saída, mas na verdade, isso faz o programa gerar pânico durante a chamada a **out.Write**:

```

if out != nil {
    out.Write([]byte("done!\n")) // pânico: desreferência de ponteiro nil
}

```

Quando **main** chama **f**, ele atribui um ponteiro **nil** do tipo ***bytes.Buffer** ao parâmetro **out**, portanto o valor dinâmico de **out** é **nil**. Apesar disso, seu tipo dinâmico é ***bytes.Buffer**, o que quer dizer que **out** é uma interface diferente de **nil** contendo um valor de ponteiro **nil** (Figura 7.5). Assim, a verificação defensiva **out != nil** continua válida.

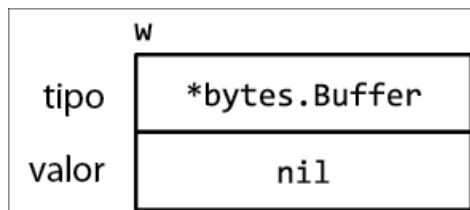


Figura 7.5 – Uma interface diferente de nil contendo um ponteiro nil.

Como antes, o sistema de despacho dinâmico determina que

`(*bytes.Buffer).Write` deve ser chamado, mas, dessa vez, com um valor de receptor que é `nil`. Para alguns tipos, como `*os.File`, `nil` é um receptor válido (seção 6.2.1), mas `*bytes.Buffer` não está entre eles. O método é chamado, mas gera pânico quando ele tenta acessar o buffer.

O problema é que, embora um ponteiro `*bytes.Buffer` `nil` tenha os métodos necessários para satisfazer a interface, ele não satisfaz os requisitos *comportamentais* da interface. Em particular, a chamada viola a pré-condição implícita de `(*bytes.Buffer).Write` de que seu receptor não seja `nil`, portanto atribuir o ponteiro `nil` à interface foi um erro. A solução é mudar o tipo de `buf` em `main` para `io.Writer`, evitando assim a atribuição do valor indevido à interface, antes de tudo:

```
var buf io.Writer
if debug {
    buf = new(bytes.Buffer) // habilita a coleta da saída
}
f(buf) // OK
```

Agora que discutimos o funcionamento dos valores interface, vamos analisar algumas das interfaces mais importantes da biblioteca-padrão de Go. Nas três próximas seções, veremos como as interfaces são usadas para ordenação, serviços web e tratamento de erros.

7.6 Ordenação com `sort.Interface`

Assim como a formatação de strings, a ordenação é uma operação usada com frequência em muitos programas. Embora um Quicksort mínimo possa ser escrito com aproximadamente 15 linhas, uma implementação robusta é bem mais extensa, e não é o tipo de código que gostaríamos de escrever de novo ou copiar sempre que precisamos dele.

Felizmente, o pacote `sort` oferece ordenação in-place para qualquer sequência, de acordo com qualquer função de ordenação. Seu design é bem incomum. Em muitas linguagens, o algoritmo de ordenação está associado ao tipo de dado da sequência, enquanto a função de ordenação está associada ao tipo dos elementos. Em comparação, a função `sort.Sort` de Go não pressupõe nada a respeito da representação da

sequência nem de seus elementos. Em vez disso, ela usa uma interface **sort.Interface** para definir o contrato entre o algoritmo genérico de ordenação e cada tipo de sequência que possa ser ordenado. Uma implementação dessa interface determina tanto a representação concreta da sequência, que com frequência é uma fatia, quanto a ordem desejada de seus elementos.

Um algoritmo de ordenação in-place precisa de três itens: o tamanho da sequência, um meio de comparar dois elementos e uma forma de trocar dois elementos (fazer swap) – portanto, esses são os três métodos de **sort.Interface**:

```
package sort

type Interface interface {
    Len() int
    Less(i, j int) bool // i, j são índices dos elementos da sequência
    Swap(i, j int)
}
```

Para ordenar qualquer sequência precisamos definir um tipo que implemente esses três métodos e, então, aplicar **sort.Sort** a uma instância desse tipo. Como talvez o exemplo mais simples possível, considere a ordenação de uma fatia de strings. O novo tipo **StringSlice** e seus métodos **Len**, **Less** e **Swap** estão a seguir:

```
type StringSlice []string

func (p StringSlice) Len() int           { return len(p) }
func (p StringSlice) Less(i, j int) bool { return p[i] < p[j] }
func (p StringSlice) Swap(i, j int)      { p[i], p[j] = p[j], p[i] }
```

Agora podemos ordenar uma fatia de strings, **names**, convertendo a fatia para um **StringSlice** assim:

```
sort.Sort(StringSlice(names))
```

A conversão produz um valor de fatia com o mesmo tamanho, capacidade e array subjacente de **names**, mas com um tipo que tem os três métodos necessários para a ordenação.

Ordenar uma fatia de strings é tão comum que o pacote **sort** oferece o tipo **StringSlice**, bem como uma função chamada **Strings**, de modo

que a chamada anterior pode ser simplificada para `sort.Strings(names)`.

A técnica nesse caso é facilmente adaptada para outras formas de ordenação, por exemplo, ignorando letras maiúsculas ou caracteres especiais. (O programa Go que ordena os termos do índice e os números de página deste livro faz isso, com uma lógica extra para algarismos romanos.) Para ordenações mais complicadas, utilizamos a mesma ideia, mas com estruturas de dados mais complexas ou implementações mais sofisticadas dos métodos de `sort.Interface`.

O exemplo de ordenação com que trabalharemos será de uma playlist de músicas, exibida como uma tabela. Cada faixa musical é uma linha única, e cada coluna é um atributo dessa faixa, como artista, título e tempo de reprodução. Suponha que uma interface gráfica de usuário apresente a tabela e que clicar no cabeçalho de uma coluna ordene a playlist de acordo com esse atributo; clicar no cabeçalho da mesma coluna novamente inverte sua ordem. Vamos verificar o que pode acontecer em resposta a cada clique.

A variável `tracks` a seguir contém uma playlist. (Um dos autores se desculpa pelo gosto musical do outro autor.) Todo elemento é indireto: é um ponteiro para um `Track`. Apesar de o código a seguir funcionar se armazenássemos os `Tracks` diretamente, a função de ordenação trocaria a posição de muitos pares de elementos; portanto, ela executará mais rápido se cada elemento for um ponteiro, que é uma única palavra de máquina, em vez de ser um `Track` completo, que pode ter oito ou mais palavras.

gopl.io/ch7/sorting

```
type Track struct {
    Title string
    Artist string
    Album string
    Year int
    Length time.Duration
}

var tracks = []*Track{
```

```

    {"Go", "Delilah", "From the Roots Up", 2012, length("3m38s")},
    {"Go", "Moby", "Moby", 1992, length("3m37s")},
    {"Go Ahead", "Alicia Keys", "As I Am", 2007, length("4m36s")},
    {"Ready 2 Go", "Martin Solveig", "Smash", 2011, length("4m24s")},
}

func length(s string) time.Duration {
    d, err := time.ParseDuration(s)
    if err != nil {
        panic(s)
    }
    return d
}

```

A função `printTracks` exibe a playlist como uma tabela. Uma apresentação gráfica seria mais bonita, mas essa pequena rotina utiliza o pacote `text/tabwriter` para gerar uma tabela cujas colunas são elegantemente alinhadas e preenchidas com espaços, como mostrado a seguir. Observe que `*tabwriter.Writer` satisfaz `io.Writer`. Ele reúne todos os dados escritos nele; seu método `Flush` formata a tabela e a escreve em `os.Stdout`.

```

func printTracks(tracks []*Track) {
    const format = "%v\t%v\t%v\t%v\t%v\t\n"
    tw := new(tabwriter.Writer).Init(os.Stdout, 0, 8, 2, ' ', 0)
    fmt.Fprintf(tw, format, "Title", "Artist", "Album", "Year", "Length")
    fmt.Fprintf(tw, format, "-----", "-----", "-----", "-----", "-----")
    for _, t := range tracks {
        fmt.Fprintf(tw, format, t.Title, t.Artist, t.Album, t.Year, t.Length)
    }
    tw.Flush() // calcula as larguras das colunas e exibe a tabela
}

```

Para ordenar a playlist de acordo com o campo `Artist`, definimos um novo tipo fatia com os métodos `Len`, `Less` e `Swap` necessários, de modo análogo ao que fizemos com `StringSlice`.

```

type byArtist []*Track

func (x byArtist) Len() int           { return len(x) }
func (x byArtist) Less(i, j int) bool { return x[i].Artist < x[j].Artist }
func (x byArtist) Swap(i, j int)      { x[i], x[j] = x[j], x[i] }

```

Para chamar a rotina genérica de ordenação, devemos inicialmente

converter **tracks** para o novo tipo **byArtist**, que define a ordem:

```
sort.Sort(byArtist(tracks))
```

Após ordenar a fatia de acordo com o artista, a saída de **printTracks** será:

Title	Artist	Album	Year	Length
-----	-----	-----	-----	-----
Go Ahead	Alicia Keys	As I Am	2007	4m36s
Go	Delilah	From the Roots Up	2012	3m38s
Ready 2 Go	Martin Solveig	Smash	2011	4m24s
Go	Moby	Moby	1992	3m37s

Se o usuário solicitar “ordenação pelo artista” uma segunda vez, as faixas serão apresentadas na ordem inversa. Porém, não precisamos definir um novo tipo **byReverseArtist** com um método **Less** invertido, pois o pacote **sort** disponibiliza uma função **Reverse** que inverte qualquer ordenação.

```
sort.Sort(sort.Reverse(byArtist(tracks)))
```

Após ordenar a fatia inversamente de acordo com o artista, a saída de **printTracks** será:

Title	Artist	Album	Year	Length
-----	-----	-----	-----	-----
Go	Moby	Moby	1992	3m37s
Ready 2 Go	Martin Solveig	Smash	2011	4m24s
Go	Delilah	From the Roots Up	2012	3m38s
Go Ahead	Alicia Keys	As I Am	2007	4m36s

A função **sort.Reverse** merece uma análise mais detalhada, pois ela usa composição (seção 6.3), que é uma ideia importante. O pacote **sort** define um tipo **reverse** não exportado, que é uma estrutura que inclui um **sort.Interface**. O método **Less** de **reverse** chama o método **Less** do valor incluído **sort.Interface**, mas com os índices ao contrário, invertendo a ordem do resultado da ordenação.

```
package sort

type reverse struct{ Interface } // isto é, sort.Interface

func (r reverse) Less(i, j int) bool { return r.Interface.Less(j, i) }
```



```
func Reverse(data Interface) Interface { return reverse{data} }
```

Len e **Swap**, que são os outros dois métodos de **reverse**, são oferecidos implicitamente pelo valor original de **sort.Interface** porque ele é um campo incluído. A função exportada **Reverse** devolve uma instância do tipo **reverse** que contém o valor original de **sort.Interface**.

Para ordenar de acordo com uma coluna diferente, devemos definir um tipo novo, por exemplo, **byYear**:

```
type byYear []*Track

func (x byYear) Len() int           { return len(x) }
func (x byYear) Less(i, j int) bool { return x[i].Year < x[j].Year }
func (x byYear) Swap(i, j int)      { x[i], x[j] = x[j], x[i] }
```

Após ordenar **tracks** de acordo com o ano usando **sort.Sort(byYear(tracks))**, **printTracks** mostra uma listagem em ordem cronológica:

Title	Artist	Album	Year	Length
-----	-----	-----	----	-----
Go	Moby	Moby	1992	3m37s
Go Ahead	Alicia Keys	As I Am	2007	4m36s
Ready 2 Go	Martin Solveig	Smash	2011	4m24s
Go	Delilah	From the Roots Up	2012	3m38s

Para todo tipo do elemento da fatia e para cada função de ordenação necessária declaramos uma nova implementação de **sort.Interface**. Como podemos ver, os métodos **Len** e **Swap** têm definições idênticas para todos os tipos de fatia. No próximo exemplo, o tipo concreto **customSort** combina uma fatia com uma função, permitindo definir uma nova ordenação escrevendo apenas a função de comparação. A propósito, os tipos concretos que implementam **sort.Interface** nem sempre são fatias; **customSort** é uma estrutura.

```
type customSort struct {
    t    []*Track
    less func(x, y *Track) bool
}

func (x customSort) Len() int           { return len(x.t) }
func (x customSort) Less(i, j int) bool { return x.less(x.t[i], x.t[j]) }
```

```
func (x customSort) Swap(i, j int)      { x.t[i], x.t[j] = x.t[j], x.t[i] }
```

Vamos definir uma função de ordenação de vários níveis (multi-tier) cuja chave principal de ordenação é **Title**, a chave secundária é **Year** e a chave terciária é a duração da música, isto é, **Length**. Aqui está a chamada a **Sort** usando uma função de ordenação anônima:

```
sort.Sort(customSort{tracks, func(x, y *Track) bool {
    if x.Title != y.Title {
        return x.Title < y.Title
    }
    if x.Year != y.Year {
        return x.Year < y.Year
    }
    if x.Length != y.Length {
        return x.Length < y.Length
    }
    return false
}})
```

E eis o resultado. Observe que o empate entre as duas faixas de nome “Go” é resolvido em favor da mais antiga.

Title	Artist	Album	Year	Length
-----	-----	-----	-----	-----
Go	Moby	Moby	1992	3m37s
Go	Delilah	From the Roots Up	2012	3m38s
Go Ahead	Alicia Keys	As I Am	2007	4m36s
Ready 2 Go	Martin Solveig	Smash	2011	4m24s

Embora ordenar uma sequência de tamanho n exija $O(n \log n)$ operações de comparação, testar se uma sequência já está ordenada exige no máximo $n-1$ comparações. A função **IsSorted** do pacote **sort** verifica isso para nós. Assim como **sort.Sort**, ela abstrai tanto a sequência quanto sua função de ordenação usando **sort.Interface**, mas ela jamais chama o método **Swap**. O código a seguir mostra as funções **IntsAreSorted** e **Ints** e o tipo **IntSlice**:

```
values := []int{3, 1, 4, 1}
fmt.Println(sort.IntsAreSorted(values)) // "false"
sort.Ints(values)
fmt.Println(values)                     // "[1 1 3 4]"
```

```
fmt.Println(sort.IntsAreSorted(values)) // "true"
sort.Sort(sort.Reverse(sort.IntSlice(values)))
fmt.Println(values)                    // "[4 3 1 1]"
fmt.Println(sort.IntsAreSorted(values)) // "false"
```

Por conveniência, o pacote **sort** oferece versões de suas funções e tipos especializados para `[]int`, `[]string` e `[]float64` usando suas ordenações naturais. Para outros tipos, como `[]int64` ou `[]uint`, dependemos de nós mesmos, embora o caminho seja curto.

Exercício 7.8: Muitas GUIs oferecem um widget de tabela com uma ordenação de vários níveis com estados (stateful multi-tier sort): a chave principal de ordenação é o cabeçalho da coluna mais recentemente clicada, a chave de ordenação secundária é o cabeçalho da segunda coluna mais recentemente clicada, e assim por diante. Defina uma implementação de **sort.Interface** para ser usada por uma tabela desse tipo. Compare essa abordagem com a ordenação repetida usando **sort.Stable**.

Exercício 7.9: Use o pacote **html/template** (seção 4.6) para substituir **printTracks** por uma função que exiba as faixas musicais como uma tabela HTML. Utilize a solução do exercício anterior para que cada clique no cabeçalho de uma coluna faça uma requisição HTTP para ordenar a tabela.

Exercício 7.10: O tipo **sort.Interface** pode ser adaptado para outros usos. Escreva uma função **IsPalindrome(s sort.Interface) bool** que informe se a sequência **s** é um palíndromo, isto é, a inversão da sequência não a torna diferente. Suponha que os elementos nos índices **i** e **j** são iguais se **s.Less(i, j) && !s.Less(j, i)**.

7.7 A interface **http.Handler**

No capítulo 1, vislumbramos o modo de usar o pacote **net/http** para implementar clientes web (seção 1.5) e servidores web (seção 1.7). Nesta seção, veremos com mais detalhes a API do servidor, cuja base é a interface **http.Handler**:

net/http

```
package http

type Handler interface {
    ServeHTTP(w ResponseWriter, r *Request)
}

func ListenAndServe(address string, h Handler) error
```

A função **ListenAndServe** exige um endereço de servidor, por exemplo, **"localhost:8000"**, e uma instância da interface **Handler** para a qual todas as requisições devem ser despachadas. Ela executa indefinidamente ou até o servidor falhar (ou falhar ao começar), devolvendo um erro que será sempre diferente de nil.

Suponha um site de e-commerce com um banco de dados que mapeia os itens à venda aos seus preços em dólares. O programa a seguir mostra a implementação mais simples que podemos imaginar. Ela modela o estoque como um tipo mapa, **database**, ao qual associamos um método **ServeHTTP** para que a interface **http.Handler** seja satisfeita. O handler percorre o mapa e exibe os itens.

gopl.io/ch7/http1

```
func main() {
    db := database{"shoes": 50, "socks": 5}
    log.Fatal(http.ListenAndServe("localhost:8000", db))
}

type dollars float32

func (d dollars) String() string { return fmt.Sprintf("%.2f", d) }

type database map[string]dollars

func (db database) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    for item, price := range db {
        fmt.Fprintf(w, "%s: %s\n", item, price)
    }
}
```

Se iniciarmos o servidor:

```
$ go build gopl.io/ch7/http1
$ ./http1 &
```

e em seguida nos conectarmos a ele com o programa **fetch** da seção 1.5

(ou com um navegador web, se você preferir), vamos obter a saída a seguir:

```
$ go build gopl.io/ch1/fetch
$ ./fetch http://localhost:8000
shoes: $50.00
socks: $5.00
```

Até agora, o servidor é capaz somente de listar seu estoque completo e fará isso para toda requisição, independentemente do URL. Um servidor mais realista define vários URLs diferentes, cada um acionando um comportamento diferente. Vamos chamar o URL existente de `/list` e acrescentar outro chamado `/price` que informe o preço de um único item, especificado como um parâmetro da requisição, como em `/price?item=socks`.

gopl.io/ch7/http2

```
func (db database) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    switch req.URL.Path {
    case "/list":
        for item, price := range db {
            fmt.Fprintf(w, "%s: %s\n", item, price)
        }
    case "/price":
        item := req.URL.Query().Get("item")
        price, ok := db[item]
        if !ok {
            w.WriteHeader(http.StatusNotFound) // 404
            fmt.Fprintf(w, "no such item: %q\n", item)
            return
        }
        fmt.Fprintf(w, "%s\n", price)
    default:
        w.WriteHeader(http.StatusNotFound) // 404
        fmt.Fprintf(w, "no such page: %s\n", req.URL)
    }
}
```

Agora o handler decide qual lógica será executada de acordo com o componente de path do URL, `req.URL.Path`. Se o handler não reconhecer o path, ele informará um erro de HTTP ao cliente chamando

`w.WriteHeader(http.StatusNotFound)`; isso deve ser feito antes de escrever qualquer texto em `w`. (A propósito, `http.ResponseWriter` é outra interface. Ela expande `io.Writer` com métodos para enviar cabeçalhos de respostas HTTP.) De modo equivalente, poderíamos ter usado a função utilitária `http.Error`:

```
msg := fmt.Sprintf("no such page: %s\n", req.URL)
http.Error(w, msg, http.StatusNotFound) // 404
```

O caso para `/price` chama o método `Query` do URL para fazer parse dos parâmetros da requisição HTTP como um mapa ou, mais precisamente, um multimapa do tipo `url.Values` (seção 6.2.1) do pacote `net/url`. Então o primeiro parâmetro `item` é encontrado, e o preço é consultado. Se o item não for encontrado, um erro será informado.

Eis uma sessão de exemplo com o novo servidor:

```
$ go build gopl.io/ch7/http2
$ go build gopl.io/ch1/fetch
$ ./http2 &
$ ./fetch http://localhost:8000/list
shoes: $50.00
socks: $5.00
$ ./fetch http://localhost:8000/price?item=socks
$5.00
$ ./fetch http://localhost:8000/price?item=shoes
$50.00
$ ./fetch http://localhost:8000/price?item=hat
no such item: "hat"
$ ./fetch http://localhost:8000/help
no such page: /help
```

Obviamente, poderíamos continuar acrescentando casos em `ServeHTTP`, mas em uma aplicação realista é conveniente definir a lógica para cada caso em uma função ou método separado. Além do mais, URLs relacionados podem precisar de uma lógica semelhante; vários arquivos de imagem podem ter URLs no formato `/images/*.png`, por exemplo. Por esses motivos, `net/http` oferece `ServeMux`, um *multiplexador de requisições* (request multiplexer) para simplificar a associação entre URLs e handlers. Um `ServeMux` agrega uma coleção de `http.Handlers` em um

único `http.Handler`. Novamente, vemos que tipos diferentes que satisfazem a mesma interface são *substituíveis*: o servidor web pode despachar requisições para qualquer `http.Handler`, independentemente de qual tipo concreto está por trás dele.

Para uma aplicação mais complexa, vários `ServeMuxes` podem ser compostos para tratar requisitos mais intrincados de despacho. Go não tem um framework web canônico, análogo ao Rails do Ruby ou ao Django de Python. Isso não quer dizer que esses frameworks não existam, porém os blocos de construção da biblioteca-padrão de Go são muito flexíveis, a ponto de frameworks não serem frequentemente necessários. Além disso, embora os frameworks sejam convenientes nas fases iniciais de um projeto, sua complexidade adicional pode dificultar a manutenção a longo prazo.

No programa a seguir, criamos um `ServeMux` e o usamos para associar os URLs aos handlers correspondentes para as operações `/list` e `/price`, que foram separadas em métodos diferentes. Então usamos o `ServeMux` como o handler principal na chamada a `ListenAndServe`.

gopl.io/ch7/http3

```
func main() {
    db := database{"shoes": 50, "socks": 5}
    mux := http.NewServeMux()
    mux.Handle("/list", http.HandlerFunc(db.list))
    mux.Handle("/price", http.HandlerFunc(db.price))
    log.Fatal(http.ListenAndServe("localhost:8000", mux))
}

type database map[string]dollars

func (db database) list(w http.ResponseWriter, req *http.Request) {
    for item, price := range db {
        fmt.Fprintf(w, "%s: %s\n", item, price)
    }
}

func (db database) price(w http.ResponseWriter, req *http.Request) {
    item := req.URL.Query().Get("item")
    price, ok := db[item]
    if !ok {
```

```

        w.WriteHeader(http.StatusNotFound) // 404
        fmt.Fprintf(w, "no such item: %q\n", item)
        return
    }
    fmt.Fprintf(w, "%s\n", price)
}

```

Vamos focar nas duas chamadas a **mux.Handle** que registram os handlers. Na primeira chamada, **db.list** é um valor de método (seção 6.4), isto é, um valor do tipo:

```
func(w http.ResponseWriter, req *http.Request)
```

Quando chamado, ele invoca o método **database.list** com o valor de receptor **db**. Sendo assim, **db.list** é uma função que implementa um comportamento semelhante ao de handler, mas como não tem métodos, ele não satisfaz a interface **http.Handler** e não pode ser passado diretamente para **mux.Handle**.

A expressão **http.HandlerFunc(db.list)** é uma conversão, e não uma chamada de função, pois **http.HandlerFunc** é um tipo. Ele tem a seguinte definição:

net/http

```

package http

type HandlerFunc func(w ResponseWriter, r *Request)

func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
    f(w, r)
}

```

HandlerFunc mostra alguns recursos incomuns do sistema de interface de Go. É um tipo função que tem métodos e satisfaz uma interface, **http.Handler**. O comportamento de seu método **ServeHTTP** é chamar a função subjacente. Desse modo, **HandlerFunc** é um adaptador que permite que um valor de função satisfaça uma interface, em que a função e o único método da interface têm a mesma assinatura. Com efeito, esse truque permite que um único tipo como **database** satisfaça a interface **http.Handler** de várias maneiras diferentes: uma vez por meio de seu método **list**, outra por meio de seu método **price**, e assim por diante.

Como registrar um handler dessa maneira é muito comum, **ServeMux** tem um método conveniente chamado **HandleFunc** que faz isso para nós, portanto podemos simplificar o código de registro de handler assim:

gopl.io/ch7/http3a

```
mux.HandleFunc("/list", db.list)
mux.HandleFunc("/price", db.price)
```

A partir do código anterior, é fácil ver como alguém poderia criar um programa em que haja dois servidores web diferentes ouvindo em portas diferentes, definindo URLs diferentes e despachando para handlers diferentes. Simplesmente criaríamos outro **ServeMux** e faríamos outra chamada para **ListenAndServe**, talvez de modo concorrente. Porém, na maioria dos programas, um servidor web será suficiente. Além disso, é comum definir handlers HTTP em vários arquivos de uma aplicação, e seria incômodo se todos eles tivessem de ser explicitamente registrados junto à instância de **ServeMux** da aplicação.

Assim, por questões de conveniência, **net/http** oferece uma instância global de **ServeMux** chamada **DefaultServeMux** e funções de nível de pacote chamadas **http.Handle** e **http.HandleFunc**. Para usar **DefaultServeMux** como o handler principal do servidor, não precisamos passá-lo para **ListenAndServe**; **nil** funcionará.

A função principal do servidor pode então ser simplificada para:

gopl.io/ch7/http4

```
func main() {
    db := database{"shoes": 50, "socks": 5}
    http.HandleFunc("/list", db.list)
    http.HandleFunc("/price", db.price)
    log.Fatal(http.ListenAndServe("localhost:8000", nil))
}
```

Por fim, um lembrete importante: conforme mencionamos na seção 1.7, o servidor web chama cada handler em uma nova gorrotina; portanto, handlers devem tomar certos cuidados como fazer o *travamento* (locking) quando acessar variáveis que outras gorrotinas, incluindo outras requisições ao mesmo handler, possam estar acessando. Falaremos mais

sobre concorrência nos próximos dois capítulos.

Exercício 7.11: Acrescente handlers adicionais para que os clientes possam criar, ler, atualizar e apagar entradas do banco de dados. Por exemplo, uma requisição no formato `/update?item=socks&price=6` atualizará o preço de um item do estoque e informará um erro se o item não existir ou se o preço for inválido. (Aviso: essa alteração introduz atualizações concorrentes em variáveis.)

Exercício 7.12: Mude o handler de `/list` para que exiba sua saída como uma tabela HTML, e não como texto. Talvez você ache o pacote `html/template` (seção 4.6) útil.

7.8 Interface error

Desde o início deste livro, usamos e criamos valores do misterioso tipo pré-declarado `error`, sem explicar o que ele realmente é. De fato, ele é apenas um tipo interface com um único método que devolve uma mensagem de erro:

```
type error interface {  
    Error() string  
}
```

A maneira mais simples de criar um erro é chamar `errors.New`, que devolve um novo `error` para uma dada mensagem de erro. O pacote `errors` todo tem apenas quatro linhas:

```
package errors  
  
func New(text string) error { return &errorString{text} }  
  
type errorString struct { text string }  
  
func (e *errorString) Error() string { return e.text }
```

O tipo subjacente de `errorString` é uma estrutura, e não uma string, para proteger sua representação de atualizações acidentais (ou premeditadas). E o motivo para o tipo ponteiro `*errorString`, e não `errorString` sozinho, satisfazer a interface `error` é que toda chamada a `New` aloca uma instância distinta de `error` que não é igual a nenhuma outra. Não queremos que um erro distinto como `io.EOF` seja comparado

como igual a um erro que, por acaso, tenha a mesma mensagem.

```
fmt.Println(errors.New("EOF") == errors.New("EOF")) // "false"
```

Chamadas a **errors.New** são relativamente raras, pois há uma função wrapper conveniente, **fmt.Errorf**, que também faz formatação de strings. Nós a usamos várias vezes no capítulo 5.

```
package fmt

import "errors"

func Errorf(format string, args ...interface{}) error {
    return errors.New(Sprintf(format, args...))
}
```

Embora ***errorString** possa ser o tipo mais simples de **error**, ele está longe de ser o único. Por exemplo, o pacote **syscall** oferece a API para chamadas de sistema de baixo nível de Go. Em muitas plataformas, ele define um tipo numérico **Errno** que satisfaz **error** e, em plataformas Unix, o método **Error** de **Errno** faz uma busca em uma tabela de strings, como mostrado a seguir:

```
package syscall

type Errno uintptr // código de erro do sistema operacional

var errors = [...]string{
    1: "operation not permitted", // EPERM
    2: "no such file or directory", // ENOENT
    3: "no such process",          // ESRCH
    // ...
}

func (e Errno) Error() string {
    if 0 <= int(e) && int(e) < len(errors) {
        return errors[e]
    }
    return fmt.Sprintf("errno %d", e)
}
```

A instrução a seguir cria um valor interface que armazena o valor 2 de **Errno**, representando a condição **ENOENT** em POSIX:

```
var err error = syscall.Errno(2)
fmt.Println(err.Error()) // "no such file or directory"
fmt.Println(err)         // "no such file or directory"
```

O valor de **err** pode ser visto graficamente na figura 7.6.

Errno é uma representação eficiente de erros de chamadas de sistema extraídos de um conjunto finito e satisfaz a interface **error** padrão. Veremos outros tipos que satisfazem essa interface na seção 7.11.

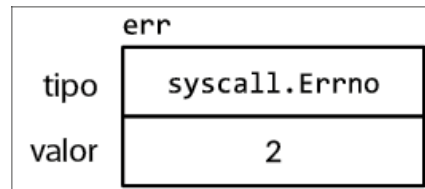


Figura 7.6 – Um valor interface que contém um inteiro `syscall.Errno`.

7.9 Exemplo: avaliador de expressões

Nesta seção, criaremos um avaliador para expressões aritméticas simples. Usaremos uma interface **Expr** para representar qualquer expressão nessa linguagem. Por enquanto, essa interface não precisa de métodos, mas acrescentaremos alguns posteriormente.

```
// Uma Expr é uma expressão aritmética.  
type Expr interface{}
```

Nossa linguagem de expressão é constituída de números de ponto flutuante literais; os operadores binários **+**, **-**, ***** e **/**; os operadores unários **-x** e **+x**; as chamadas de função **pow(x,y)**, **sin(x)** e **sqrt(x)**; variáveis como **x** e **pi**; e, é claro, parênteses e precedência de operadores padrão. Todos os valores são do tipo **float64**. Eis alguns exemplos de expressões:

```
sqrt(A / pi)  
pow(x, 3) + pow(y, 3)  
(F - 32) * 5 / 9
```

Os cinco tipos concretos a seguir representam tipos específicos de expressão. Um **Var** representa uma referência a uma variável. (Veremos em breve por que ele é exportado.) Um **literal** representa uma constante de ponto flutuante. Os tipos **unary** e **binary** representam expressões de operadores com um ou dois operandos, que podem ser qualquer tipo de **Expr**. Um **call** representa uma chamada de função; restringiremos seu campo **fn** a **pow**, **sin** ou **sqrt**.

gopl.io/ch7/eval

```
// Um Var identifica uma variável, por exemplo, x.
type Var string

// Um literal é uma constante numérica, por exemplo, 3.141.
type literal float64

// Um unary representa uma expressão com operador unário, por exemplo, -x.
type unary struct {
    op rune // um valor entre '+', '-'
    x Expr
}

// Um binary representa uma expressão com operador binário, por exemplo, x+y.
type binary struct {
    op rune // um valor entre '+', '-', '*', '/'
    x, y Expr
}

// Um call representa uma expressão de chamada de função, por exemplo, sin(x).
type call struct {
    fn string // um valor entre "pow", "sin", "sqrt"
    args []Expr
}
```

Para avaliar uma expressão contendo variáveis, precisaremos de um *ambiente* que mapeie nomes de variáveis a valores:

```
type Env map[Var]float64
```

Também precisaremos que cada tipo de expressão defina um método **Eval** que devolva o valor da expressão em um dado ambiente. Como toda expressão deve fornecer esse método, ele será adicionado à interface **Expr**. O pacote exporta apenas os tipos **Expr**, **Env** e **Var**; clientes podem usar o avaliador sem ter acesso aos outros tipos de expressão.

```
type Expr interface {
    // Eval devolve o valor desta Expr no ambiente env.
    Eval(env Env) float64
}
```

Os métodos **Eval** concretos estão mostrados a seguir. O método para **Var** executa uma busca no ambiente, que devolve zero se a variável não estiver definida, e o método para **literal** simplesmente devolve o valor literal.

```
func (v Var) Eval(env Env) float64 {
```

```

    return env[v]
}

func (l literal) Eval(_ Env) float64 {
    return float64(l)
}

```

Os métodos `Eval` para `unary` e `binary` avaliam recursivamente seus operandos e, em seguida, aplicam a operação `op` a eles. Não consideramos divisões por zero nem por infinito como erros, pois eles geram um resultado, apesar de não serem finitos. Por fim, o método para `call` avalia os argumentos da função `pow`, `sin` ou `sqrt` e, então, chama a função correspondente do pacote `math`.

```

func (u unary) Eval(env Env) float64 {
    switch u.op {
    case '+':
        return +u.x.Eval(env)
    case '-':
        return -u.x.Eval(env)
    }
    panic(fmt.Sprintf("unsupported unary operator: %q", u.op))
}

func (b binary) Eval(env Env) float64 {
    switch b.op {
    case '+':
        return b.x.Eval(env) + b.y.Eval(env)
    case '-':
        return b.x.Eval(env) - b.y.Eval(env)
    case '*':
        return b.x.Eval(env) * b.y.Eval(env)
    case '/':
        return b.x.Eval(env) / b.y.Eval(env)
    }
    panic(fmt.Sprintf("unsupported binary operator: %q", b.op))
}

func (c call) Eval(env Env) float64 {
    switch c.fn {
    case "pow":
        return math.Pow(c.args[0].Eval(env), c.args[1].Eval(env))
    case "sin":
        return math.Sin(c.args[0].Eval(env))
    }
}

```

```

    case "sqrt":
        return math.Sqrt(c.args[0].Eval(env))
    }
    panic(fmt.Sprintf("unsupported function call: %s", c.fn))
}

```

Vários desses métodos podem falhar. Por exemplo, uma expressão com **call** poderia conter uma função desconhecida ou o número incorreto de argumentos. Também é possível criar uma expressão **unary** ou **binary** com um operador inválido como **!** ou **<** (embora a função **Parse** mencionada a seguir jamais faça isso). Esses erros fazem **Eval** gerar pânico. Outros erros, como avaliar um **Var** não presente no ambiente, simplesmente fazem **Eval** devolver o resultado incorreto. Todos esses erros podem ser detectados inspecionando **Expr** antes de avaliá-la. Esse será o trabalho do método **Check**, que mostraremos em breve, mas antes disso vamos testar **Eval**.

A função **TestEval** a seguir é um teste para o avaliador. Ela usa o pacote **testing**, que explicaremos no capítulo 11, mas por enquanto é suficiente saber que chamar **t.Errorf** informa um erro. A função percorre uma tabela de entradas que define três expressões e diferentes ambientes para cada uma, usando um loop. A primeira expressão calcula o raio de um círculo, dada sua área **A**; a segunda calcula a soma dos cubos de duas variáveis **x** e **y**; e a terceira converte uma temperatura **F** em Fahrenheit para Celsius.

```

func TestEval(t *testing.T) {
    tests := []struct {
        expr string
        env Env
        want string
    }{
        {"sqrt(A / pi)", Env{"A": 87616, "pi": math.Pi}, "167"},
        {"pow(x, 3) + pow(y, 3)", Env{"x": 12, "y": 1}, "1729"},
        {"pow(x, 3) + pow(y, 3)", Env{"x": 9, "y": 10}, "1729"},
        {"5 / 9 * (F - 32)", Env{"F": -40}, "-40"},
        {"5 / 9 * (F - 32)", Env{"F": 32}, "0"},
        {"5 / 9 * (F - 32)", Env{"F": 212}, "100"},
    }
    var prevExpr string

```

```

for _, test := range tests {
    // Exibe expr somente quando mudar.
    if test.expr != prevExpr {
        fmt.Printf("\n%s\n", test.expr)
        prevExpr = test.expr
    }
    expr, err := Parse(test.expr)
    if err != nil {
        t.Error(err) // erro de parse
        continue
    }
    got := fmt.Sprintf("%.6g", expr.Eval(test.env))
    fmt.Printf("\t%v => %s\n", test.env, got)
    if got != test.want {
        t.Errorf("%s.Eval() in %v = %q, want %q\n",
            test.expr, test.env, got, test.want)
    }
}
}

```

Para cada entrada da tabela, o teste faz parse da expressão, avalia-a no ambiente e exibe o resultado. Não temos espaço para mostrar a função **Parse** aqui, mas você a encontrará se fizer download do pacote usando **go get**.

O comando **go test** (seção 11.1) executa os testes de um pacote:

```
$ go test -v gopl.io/ch7/eval
```

A flag **-v** permite ver a saída do teste, que em geral é suprimida em um teste bem-sucedido como esse. Eis a saída das instruções **fmt.Printf** do teste:

```

sqrt(A / pi)
map[A:87616 pi:3.141592653589793] => 167

pow(x, 3) + pow(y, 3)
map[x:12 y:1] => 1729
map[x:9 y:10] => 1729

5 / 9 * (F - 32)
map[F:-40] => -40
map[F:32] => 0
map[F:212] => 100

```

Felizmente, as entradas até agora foram todas bem formadas, mas é

improvável que nossa sorte dure. Mesmo em linguagens interpretadas, é comum verificar a sintaxe para erros *estáticos*, ou seja, erros que podem ser detectados sem executar o programa. Ao separar as verificações estáticas das verificações dinâmicas, podemos detectar erros com antecedência e fazer várias verificações somente uma vez, e não a cada vez que uma expressão é avaliada.

Vamos acrescentar outro método à interface **Expr**. O método **Check** verifica erros estáticos na árvore sintática de uma expressão. Explicaremos seu parâmetro **vars** em breve.

```
type Expr interface {
    Eval(env Env) float64
    // Check informa erros nessa Expr e adiciona seus Vars ao conjunto.
    Check(vars map[Var]bool) error
}
```

Os métodos **Check** concretos estão mostrados a seguir. A avaliação de **literal** e de **Var** não pode falhar, portanto os métodos **Check** para esses tipos devolvem **nil**. Os métodos para **unary** e **binary** inicialmente conferem se o operador é válido e, então, verificam de modo recursivo os operandos. De forma semelhante, o método para **call** verifica se a função é conhecida e se tem o número correto de argumentos e, então, confere recursivamente cada argumento.

```
func (v Var) Check(vars map[Var]bool) error {
    vars[v] = true
    return nil
}

func (literal) Check(vars map[Var]bool) error {
    return nil
}

func (u unary) Check(vars map[Var]bool) error {
    if !strings.ContainsRune("+-", u.op) {
        return fmt.Errorf("unexpected unary op %q", u.op)
    }
    return u.x.Check(vars)
}

func (b binary) Check(vars map[Var]bool) error {
    if !strings.ContainsRune("+-*/", b.op) {
```

```

        return fmt.Errorf("unexpected binary op %q", b.op)
    }
    if err := b.x.Check(vars); err != nil {
        return err
    }
    return b.y.Check(vars)
}

func (c call) Check(vars map[Var]bool) error {
    arity, ok := numParams[c.fn]
    if !ok {
        return fmt.Errorf("unknown function %q", c.fn)
    }
    if len(c.args) != arity {
        return fmt.Errorf("call to %s has %d args, want %d",
            c.fn, len(c.args), arity)
    }
    for _, arg := range c.args {
        if err := arg.Check(vars); err != nil {
            return err
        }
    }
    return nil
}

var numParams = map[string]int{"pow": 2, "sin": 1, "sqrt": 1}

```

Listamos um conjunto de entradas com problema e os erros que elas geram em dois grupos. A função **Parse** (não foi mostrada) informa erros de sintaxe e a função **Check** informa erros semânticos.

x % 2	unexpected '%'
math.Pi	unexpected '.'
!true	unexpected '!'
"hello"	unexpected '''
log(10)	unknown function "log"
sqrt(1, 2)	call to sqrt has 2 args, want 1

O argumento de **Check**, que é um conjunto de **Vars**, armazena o conjunto de nomes de variáveis encontrados na expressão. Cada uma dessas variáveis deve estar presente no ambiente para a avaliação ter sucesso. Esse conjunto é logicamente o *resultado* da chamada a **Check**, mas, como o método é recursivo, é mais conveniente que **Check** preencha um conjunto

passado como parâmetro. O cliente deve fornecer um conjunto vazio na chamada inicial.

Na seção 3.2, desenhamos uma função $f(x,y)$ fixa em tempo de compilação. Agora que podemos fazer parse, verificar e avaliar expressões em strings, podemos criar uma aplicação web que receba uma expressão do cliente em tempo de execução e desenhar a superfície dessa função. Podemos usar o conjunto **vars** para verificar se a expressão é uma função com apenas duas variáveis, **x** e **y** – três, na verdade, pois forneceremos **r**, que é o raio, por conveniência. Além disso, usaremos o método **Check** para rejeitar expressões malformadas antes de a avaliação começar, para que não precisemos repetir essas verificações durante as 40 mil avaliações (100 x 100 células, cada uma com quatro cantos) da função que se segue.

A função **parseAndCheck** combina esses passos de parsing e de verificação:

gopl.io/ch7/surface

```
import "gopl.io/ch7/eval"

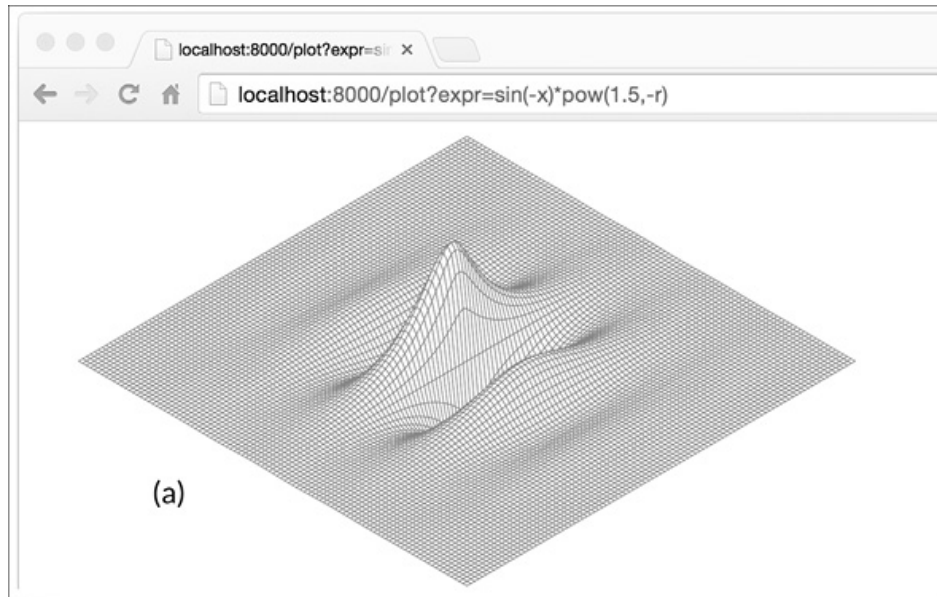
func parseAndCheck(s string) (eval.Expr, error) {
    if s == "" {
        return nil, fmt.Errorf("empty expression")
    }
    expr, err := eval.Parse(s)
    if err != nil {
        return nil, err
    }
    vars := make(map[eval.Var]bool)
    if err := expr.Check(vars); err != nil {
        return nil, err
    }
    for v := range vars {
        if v != "x" && v != "y" && v != "r" {
            return nil, fmt.Errorf("undefined variable: %s", v)
        }
    }
    return expr, nil
}
```

Para fazer desse código uma aplicação web, tudo de que precisamos é da

função **plot** a seguir, que tem a conhecida assinatura de um **http.HandlerFunc**:

```
func plot(w http.ResponseWriter, r *http.Request) {
    r.ParseForm()
    expr, err := parseAndCheck(r.Form.Get("expr"))
    if err != nil {
        http.Error(w, "bad expr: "+err.Error(), http.StatusBadRequest)
        return
    }
    w.Header().Set("Content-Type", "image/svg+xml")
    surface(w, func(x, y float64) float64 {
        r := math.Hypot(x, y) // distância de (0,0)
        return expr.Eval(eval.Env{"x": x, "y": y, "r": r})
    })
}
```

A função **plot** faz parse e verifica a expressão especificada na requisição HTTP e a usa para criar uma função anônima de duas variáveis. A função anônima tem a mesma assinatura da função fixa **f** do programa original de desenho de superfície, mas ela avalia a expressão fornecida pelo usuário. O ambiente define **x**, **y** e o raio **r**. Por fim, **plot** chama **surface**, que é simplesmente a função **main** de **gopl.io/ch3/surface**, modificada para aceitar a função de desenho e a saída **io.Writer** como parâmetros, em vez de usar a função fixa **f** e **os.Stdout**. A figura 7.7 mostra três superfícies geradas pelo programa.



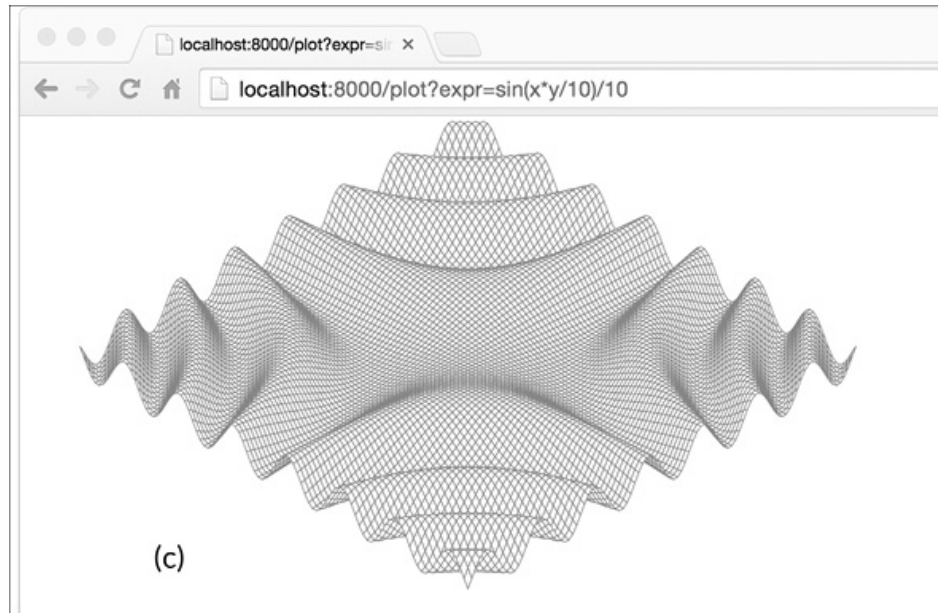


Figura 7.7 – As superfícies de três funções: (a) $\sin(-x) \cdot \text{pow}(1.5, -r)$; (b) $\text{pow}(2, \sin(y)) \cdot \text{pow}(2, \sin(x)) / 12$; (c) $\sin(x \cdot y / 10) / 10$.

Exercício 7.13: Acrescente um método `String` a `Expr` para um pretty-print da árvore sintática. Verifique se os resultados, quando submetidos novamente ao parsing, produzem uma árvore equivalente.

Exercício 7.14: Defina um novo tipo concreto que satisfaça a interface `Expr` e ofereça uma nova operação, como calcular o valor mínimo de seus operandos. Como a função `Parse` não cria instâncias desse tipo novo, para usá-la você precisará construir diretamente uma árvore sintática (ou estender o parser).

Exercício 7.15: Escreva um programa que leia uma única expressão da entrada-padrão, peça que o usuário forneça valores para quaisquer variáveis e, então, avalie a expressão no ambiente resultante. Trate todos os erros com elegância.

Exercício 7.16: Escreva um programa de calculadora baseado em web.

7.10 Asserções de tipo

Uma *asserção de tipo* (type assertion) é uma operação aplicada a um valor interface. Sintaticamente, ela é semelhante a `x.(T)`, em que `x` é uma

expressão de um tipo interface e *T* é um tipo, chamado tipo “asserido” (asserted). Uma asserção de tipo verifica se o tipo dinâmico de seu operando coincide com o tipo asserido.

Há duas possibilidades. Em primeiro lugar, se o tipo asserido *T* for concreto, a asserção de tipo verifica se o tipo dinâmico de *x* é *idêntico a T*. Se essa verificação for bem-sucedida, o resultado da asserção de tipo é o valor dinâmico de *x*, cujo tipo, é claro, é *T*. Em outras palavras, uma asserção de tipo para um tipo concreto extrai o valor concreto de seu operando. Se a verificação falhar, a operação gera pânico. Por exemplo:

```
var w io.Writer
w = os.Stdout
f := w.(*os.File)      // sucesso: f == os.Stdout
c := w.(*bytes.Buffer) // pânico: a interface contém *os.File,
                        // e não *bytes.Buffer
```

Em segundo lugar, se o tipo asserido *T* for um tipo interface, a asserção de tipo verifica se o tipo dinâmico de *x* *satisfaz T*. Se essa verificação for bem-sucedida, o valor dinâmico não é extraído; o resultado continua sendo um valor interface com os mesmos componentes de tipo e valor, mas o resultado tem o tipo de interface *T*. Em outras palavras, uma asserção de tipo para um tipo interface altera o tipo da expressão, deixando um conjunto diferente (e geralmente maior) de métodos acessível, mas preserva os componentes de tipo e valor dinâmicos no valor da interface.

Após a primeira asserção de tipo a seguir, tanto *w* quanto *rw* contêm **os.Stdout**, portanto cada um tem um tipo dinâmico igual a ***os.File**; mas *w*, que é um **io.Writer**, expõe apenas o método **Write** de arquivo, enquanto *rw* expõe seu método **Read** também.

```
var w io.Writer
w = os.Stdout
rw := w.(io.ReadWriter) // sucesso: *os.File tem tanto Read quanto Write

w = new(ByteCounter)
rw = w.(io.ReadWriter)  // pânico: *ByteCounter não tem um método Read
```

Independentemente do tipo asserido, se o operando for um valor interface nil, a asserção de tipo falhará. Uma asserção de tipo com um tipo interface menos restritivo (com menos métodos) raramente é necessária, pois ela se

comporta como uma atribuição, exceto no caso de `nil`.

```
w = rw // io.ReadWriter pode ser atribuído a io.Writer
w = rw.(io.Writer) // falha somente se rw == nil
```

Muitas vezes não temos certeza do tipo dinâmico de um valor interface e gostaríamos de testar se ele é de algum tipo em particular. Se a asserção de tipo aparecer em uma atribuição em que dois resultados são esperados, como nas declarações a seguir, a operação não gera pânico em caso de falha, mas devolve um segundo resultado adicional, isto é, um booleano que indica sucesso:

```
var w io.Writer = os.Stdout
f, ok := w.(*os.File) // sucesso: ok, f == os.Stdout
b, ok := w.(*bytes.Buffer) // falha: !ok, b == nil
```

O segundo resultado, por convenção, é atribuído a uma variável chamada `ok`. Se a operação falhar, `ok` é falso, e o primeiro resultado é igual ao valor zero do tipo asserido que, nesse exemplo, é um `*bytes.Buffer` `nil`.

O resultado `ok` geralmente é usado imediatamente para decidir o que deve ser feito a seguir. A forma estendida da instrução `if` torna isso bem compacto:

```
if f, ok := w.(*os.File); ok {
    // ...usa f...
}
```

Quando o operando de uma asserção de tipo é uma variável, em vez de inventar outro nome para a nova variável local, às vezes você verá o nome original ser reutilizado, encobrendo o original, desta maneira:

```
if w, ok := w.(*os.File); ok {
    // ...usa w...
}
```

7.11 Diferenciando erros com asserções de tipo

Considere o conjunto de erros devolvido por operações de arquivo no pacote `os`. A E/S pode apresentar falhas por muitos motivos, mas três tipos de falha com frequência devem ser tratados de modo diferente: arquivo já existente (para operações de criação), arquivo não encontrado

(para operações de leitura) e permissão não concedida. O pacote **os** oferece estas três funções auxiliares para classificar a falha informada por um dado valor de **error**:

```
package os

func IsExist(err error) bool
func IsNotExist(err error) bool
func IsPermission(err error) bool
```

Uma implementação ingênua de uma dessas funções pode verificar se a mensagem de erro contém uma determinada substring:

```
func IsNotExist(err error) bool {
    // NOTA: não é robusto!
    return strings.Contains(err.Error(), "file does not exist")
}
```

mas, como a lógica para tratamento de erros de E/S pode variar de uma plataforma para outra, essa abordagem não é robusta, e a mesma falha pode ser informada com várias mensagens de erro diferentes. Verificar substrings em mensagens de erro pode ser útil durante os testes para garantir que as funções falham da maneira esperada, porém é inadequado para código de produção.

Uma abordagem mais confiável é representar valores de erro estruturados usando um tipo dedicado. O pacote **os** define um tipo chamado **PathError** para descrever falhas que envolvam uma operação em um path de arquivo, como **Open** ou **Delete**, e uma variante chamada **LinkError** para descrever falhas de operações que envolvam dois paths de arquivo, como **Symlink** e **Rename**. Eis o código de **os.PathError**:

```
package os

// PathError registra um erro, além da operação e o path de arquivo que o
// provocou.
type PathError struct {
    Op    string
    Path  string
    Err   error
}

func (e *PathError) Error() string {
    return e.Op + " " + e.Path + ": " + e.Err.Error()
}
```

```
}
```

A maioria dos clientes não toma conhecimento de **PathError** e lida com todos os erros de modo uniforme chamando seus métodos **Error**. Embora o método **Error** de **PathError** forme uma mensagem simplesmente concatenando os campos, a estrutura de **PathError** preserva os componentes subjacentes do erro. Clientes que precisam distinguir um tipo de falha de outro podem usar uma asserção de tipo para identificar o tipo específico do erro, o qual oferece mais detalhes que uma simples string.

```
_, err := os.Open("/no/such/file")
fmt.Println(err) // "open /no/such/file: No such file or directory"
fmt.Printf("%#v\n", err)
// Saída:
// &os.PathError{Op:"open", Path:"/no/such/file", Err:0x2}
```

É assim que as três funções auxiliares operam. Por exemplo, **IsNotExist**, mostrada a seguir, informa se um erro é igual a **syscall.ENOENT** (seção 7.8), ou ao erro diferenciado **os.ErrNotExist** (veja **io.EOF** na seção 5.4.2), ou se é um ***PathError**, cujo erro subjacente é um desses dois erros.

```
import (
    "errors"
    "syscall"
)

var ErrNotExist = errors.New("file does not exist")

// IsNotExist devolve um booleano informando se o erro indica
// que um arquivo ou diretório não existe. Ele é satisfeito por
// ErrNotExist assim como por alguns erros de syscall.
func IsNotExist(err error) bool {
    if pe, ok := err.(*PathError); ok {
        err = pe.Err
    }
    return err == syscall.ENOENT || err == ErrNotExist
}
```

E aqui está ela em ação:

```
_, err := os.Open("/no/such/file")
fmt.Println(os.IsNotExist(err)) // "true"
```

É claro que a estrutura de **PathError** será perdida se a mensagem de erro

for combinada em uma string maior, por exemplo, por uma chamada a `fmt.Errorf`. A discriminação de erros normalmente deve ser feita logo após a operação de falha, antes de um erro ser propagado a quem fez a chamada.

7.12 Consultando comportamentos com asserções de tipo interface

A lógica a seguir é semelhante à parte do servidor web de `net/http` responsável por escrever campos do cabeçalho HTTP, como `"Content-type: text/html"`. A resposta HTTP é representada por `io.Writer w`; os bytes escritos nela, em última instância, são enviados ao navegador web de alguém.

```
func writeHeader(w io.Writer, contentType string) error {
    if _, err := w.Write([]byte("Content-Type: ")); err != nil {
        return err
    }
    if _, err := w.Write([]byte(contentType)); err != nil {
        return err
    }
    // ...
}
```

Como o método `Write` exige uma fatia de bytes e o valor que queremos escrever é uma string, uma conversão `[]byte(...)` é necessária. Essa conversão aloca memória e faz uma cópia, mas esta é descartada quase imediatamente na sequência. Vamos supor que essa seja uma parte essencial do servidor web e que nosso gerador de perfil (profiling) revelou que a alocação de memória está deixando o servidor lento. É possível evitar a alocação de memória aqui?

A interface `io.Writer` informa apenas um fato sobre o tipo concreto que `w` armazena: que bytes podem ser escritos nele. Se espiarmos por trás das cortinas do pacote `net/http`, veremos que o tipo dinâmico que `w` armazena nesse programa também tem um método `WriteString` que permite que strings sejam eficientemente escritas nele, evitando a

necessidade de alocar uma cópia temporária. (Isso pode parecer um tiro no escuro, mas alguns tipos importantes que satisfazem `io.Writer` também têm um método `WriteString`, incluindo `*bytes.Buffer`, `*os.File` e `*bufio.Writer`.)

Não podemos pressupor que um `io.Writer` `w` qualquer também tenha o método `WriteString`. Porém, podemos definir uma nova interface que tenha apenas esse método e usar uma asserção de tipo para testar se o tipo dinâmico de `w` satisfaz essa nova interface.

```
// writeString escreve s em w.
// Se w tem um método WriteString, ele é chamado no lugar de w.Write.
func writeString(w io.Writer, s string) (n int, err error) {
    type stringWriter interface {
        WriteString(string) (n int, err error)
    }
    if sw, ok := w.(stringWriter); ok {
        return sw.WriteString(s) // evita uma cópia
    }
    return w.Write([]byte(s)) // aloca uma cópia temporária
}

func writeHeader(w io.Writer, contentType string) error {
    if _, err := writeString(w, "Content-Type: "); err != nil {
        return err
    }
    if _, err := writeString(w, contentType); err != nil {
        return err
    }
    // ...
}
```

Para evitar repetição, transferimos a verificação para a função utilitária `writeString`, mas ela é tão útil que a biblioteca-padrão a disponibiliza como `io.WriteString`. É a maneira recomendada de escrever uma string em um `io.Writer`.

O curioso nesse exemplo é que não há uma interface padrão que defina o método `WriteString` e especifique seu comportamento necessário. Além do mais, o fato de um tipo concreto satisfazer ou não a interface `stringWriter` é determinado somente por seus métodos, e não por

qualquer relacionamento declarado entre ele e o tipo interface. Isso quer dizer que a técnica anterior conta com a suposição de que se um tipo satisfaz a interface a seguir, *então* `WriteString(s)` deve ter o mesmo efeito que `Write([]byte(s))`.

```
interface {  
    io.Writer  
    WriteString(s string) (n int, err error)  
}
```

Embora `io.WriteString` documente sua suposição, poucas funções que a chamam provavelmente documentarão que também assumem esse pressuposto. Definir um método de um tipo particular é tomado como uma concordância implícita para um determinado contrato comportamental. As pessoas para quem Go é novidade, em especial aquelas com um background em linguagens fortemente tipadas, podem achar que essa falta de intenção explícita é perturbadora, mas na prática raramente é um problema. Com exceção da interface vazia `interface{}`, tipos interface raramente são satisfeitos por coincidência.

A função `writeString` anterior usa uma asserção de tipo para ver se um valor de um tipo interface genérico também satisfaz um tipo interface mais específico e, em caso afirmativo, utiliza os comportamentos da interface específica. Essa técnica pode ter bom uso, independentemente de a interface consultada ser padrão, como `io.ReadWriter`, ou definida pelo usuário, como `stringWriter`.

É assim também que `fmt.Fprintf` distingue valores que satisfazem `error` ou `fmt.Stringer` de todos os outros valores. Em `fmt.Fprintf`, há um passo que converte um único operando em uma string – algo como:

```
package fmt  
  
func formatOneValue(x interface{}) string {  
    if err, ok := x.(error); ok {  
        return err.Error()  
    }  
    if str, ok := x.(Stringer); ok {  
        return str.String()  
    }  
    // ...todos os demais tipos...
```

}

Se `x` satisfaz uma das duas interfaces, isso determina a formatação do valor. Se não satisfaz, o caso default trata todos os outros tipos de forma mais ou menos uniforme usando reflexão; veremos como isso se dá no capítulo 12.

Novamente, esse código supõe que qualquer tipo com um método `String` satisfaz o contrato comportamental de `fmt.Stringer`, que é devolver uma string adequada para exibição.

7.13 Switches de tipo

Interfaces são usadas em dois estilos distintos. No primeiro estilo, exemplificado por `io.Reader`, `io.Writer`, `fmt.Stringer`, `sort.Interface`, `http.Handler` e `error`, os métodos de uma interface expressam as semelhanças entre os tipos concretos que satisfazem a interface, mas ocultam os detalhes de representação e as operações intrínsecas desses tipos concretos. A ênfase está nos métodos, e não nos tipos concretos.

O segundo estilo aproveita a capacidade de um valor interface de armazenar valores de uma variedade de tipos concretos e considera a interface como a *união* desses tipos. As asserções de tipo são usadas para fazer dinamicamente a distinção entre esses tipos e tratar cada caso de modo diferente. Nesse estilo, a ênfase está nos tipos concretos que satisfazem a interface, e não nos métodos da interface (se é que ela tem algum), e não há ocultação de informação. Descreveremos as interfaces usadas dessa maneira como *uniões discriminadas* (discriminated unions).

Se você tem familiaridade com programação orientada a objetos, talvez reconheça esses dois estilos como *polimorfismo de subtipo* e *polimorfismo ad hoc*, mas não é preciso se lembrar desses termos. No restante deste capítulo apresentaremos exemplos do segundo estilo.

A API de Go para consultar um banco de dados SQL, como aquelas de outras linguagens, permite separar claramente a parte fixa de uma query das partes variáveis. Um exemplo de cliente pode ter o seguinte aspecto:

```
import "database/sql"

func listTracks(db sql.DB, artist string, minYear, maxYear int) {
    result, err := db.Exec(
        "SELECT * FROM tracks WHERE artist = ? AND ? <= year AND year <= ?",
        artist, minYear, maxYear)
    // ...
}
```

O método **Exec** substitui cada '?' da string de query por um SQL literal que representa o valor do argumento correspondente; ele pode ser um booleano, um número, uma string ou **nil**. Compor queries dessa maneira pode ajudar a evitar ataques de injeção de SQL, em que um inimigo assume o controle da query explorando o uso indevido de aspas nos dados de entrada. Em **Exec**, podemos encontrar uma função como a que se segue, que converte o valor de cada argumento em sua notação SQL literal.

```
func sqlQuote(x interface{}) string {
    if x == nil {
        return "NULL"
    } else if _, ok := x.(int); ok {
        return fmt.Sprintf("%d", x)
    } else if _, ok := x.(uint); ok {
        return fmt.Sprintf("%d", x)
    } else if b, ok := x.(bool); ok {
        if b {
            return "TRUE"
        }
        return "FALSE"
    } else if s, ok := x.(string); ok {
        return sqlQuoteString(s) // (não está sendo mostrado)
    } else {
        panic(fmt.Sprintf("unexpected type %T: %v", x, x))
    }
}
```

Uma instrução **switch** simplifica uma cadeia de **if-else** que execute uma série de testes de igualdade de valores. Uma instrução análoga de *switch de tipo* simplifica uma cadeia de **if-else** de asserções de tipo.

Em sua forma mais simples, um switch de tipo parece uma instrução switch comum, mas o operando é **x.(type)** – é literalmente a palavra

reservada **type** –, e cada caso tem um ou mais tipos. Um switch de tipo permite uma ramificação múltipla baseada no tipo dinâmico do valor da interface. O caso **nil** corresponde a **x == nil** e o caso **default** inclui quaisquer casos para os quais não há correspondência. Um switch de tipo para **sqlQuote** teria os casos a seguir:

```
switch x.(type) {  
    case nil:          // ...  
    case int, uint:    // ...  
    case bool:         // ...  
    case string:       // ...  
    default:           // ...  
}
```

Como em uma instrução switch comum (seção 1.8), os casos são considerados na sequência e, quando há uma correspondência, o corpo do caso é executado. A ordem dos casos torna-se significativa quando um ou mais tipos são interfaces, o que possibilita haver correspondência com dois casos. A posição do caso **default** em relação aos demais não importa. Um **fallthrough** não é permitido.

Observe que, na função original, a lógica para os casos **bool** e **string** precisa ter acesso ao valor extraído pela asserção de tipo. Como isso é comum, a instrução de switch de tipo tem uma forma estendida que vincula o valor extraído a uma nova variável em cada caso:

```
switch x := x.(type) { /* ... */ }
```

Aqui também chamamos as novas variáveis de **x**; como nas asserções de tipo, reutilizar nomes de variáveis é comum. Assim como em uma instrução **switch**, um switch de tipo cria implicitamente um bloco léxico, portanto a declaração da nova variável chamada **x** não entra em conflito com uma variável **x** em um bloco mais externo. Cada **case** também cria implicitamente um bloco léxico separado.

Reescrever **sqlQuote** de modo a usar a forma estendida do switch de tipo deixa-a significativamente mais clara:

```
func sqlQuote(x interface{}) string {  
    switch x := x.(type) {  
    case nil:
```



```

    return "NULL"
case int, uint:
    return fmt.Sprintf("%d", x) // x tem tipo interface{} aqui.
case bool:
    if x {
        return "TRUE"
    }
    return "FALSE"
case string:
    return sqlQuoteString(x) // (não está sendo mostrado)
default:
    panic(fmt.Sprintf("unexpected type %T: %v", x, x))
}
}

```

Nessa versão, no bloco de cada caso que tem um único tipo, a variável `x` tem o mesmo tipo do caso. Por exemplo, `x` tem tipo `bool` no caso para `bool` e tipo `string` no caso para `string`. Em todos os demais casos, `x` tem o tipo (interface) do operando de `switch`, que é `interface{}` nesse exemplo. Quando a mesma ação é necessária em vários casos, como `int` e `uint`, o `switch` de tipo facilita combiná-los.

Embora `sqlQuote` aceite um argumento de qualquer tipo, a função executa até o fim somente se o tipo do argumento coincidir com um dos casos do `switch` de tipo; caso contrário, um pânico é gerado com uma mensagem de “tipo não esperado”. Embora o tipo de `x` seja `interface{}`, o consideramos como uma *união discriminada* de `int`, `uint`, `bool`, `string` e `nil`.

7.14 Exemplo: decodificação de XML baseada em token

A seção 4.5 mostrou como decodificar documentos JSON em estruturas de dados Go com as funções `Marshal` e `Unmarshal` do pacote `encoding/json`. O pacote `encoding/xml` oferece uma API semelhante. Essa abordagem é conveniente quando queremos construir uma representação da árvore do documento, mas isso não é necessário em muitos programas. O pacote `encoding/xml` também oferece uma API de baixo nível *baseada em tokens* para decodificação de XML. No estilo

baseado em token, o parser consome a entrada e produz um stream de tokens, principalmente de quatro tipos – **StartElement**, **EndElement**, **CharData** e **Comment** –, em que cada um é um tipo concreto no pacote `encoding/xml`. Toda chamada a `(*xml.Decoder).Token` devolve um token.

As partes relevantes da API estão a seguir:

`encoding/xml`

```
package xml

type Name struct {
    Local string // por exemplo, "Title" ou "id"
}

type Attr struct { // por exemplo, name="value"
    Name Name
    Value string
}

// Um Token inclui StartElement, EndElement, CharData
// e Comment, além de alguns tipos esotéricos (não estão sendo mostrados).
type Token interface{}
type StartElement struct { // por exemplo, <name>
    Name Name
    Attr []Attr
}

type EndElement struct { Name Name } // por exemplo, </name>
type CharData []byte                // por exemplo, <p>CharData</p>
type Comment []byte                  // por exemplo, <!-- Comment -->

type Decoder struct{ /* ... */ }
func NewDecoder(io.Reader) *Decoder
func (*Decoder) Token() (Token, error) // devolve o próximo Token na sequência
```

A interface **Token**, que não tem métodos, também é um exemplo de união discriminada. O propósito de uma interface tradicional como **io.Reader** é ocultar detalhes dos tipos concretos que a satisfazem, de modo que novas implementações possam ser criadas; todos os tipos concretos são tratados uniformemente. Em comparação, o conjunto de tipos concretos que satisfaz uma união discriminada é fixo pelo design e não é oculto, mas exposto. Tipos de união discriminada têm poucos métodos; funções

que operam neles são expressas como um conjunto de casos usando um switch de tipo, com uma lógica diferente em cada caso.

O programa **xmlselect** a seguir extrai e exibe o texto encontrado abaixo de determinados elementos em uma árvore de documento XML. Usando a API anterior, ele pode fazer sua tarefa em uma única passagem pela entrada sem sequer materializar a árvore.

gopl.io/ch7/xmlselect

```
// Xmlselect exibe o texto de elementos selecionados em um documento XML.
package main

import (
    "encoding/xml"
    "fmt"
    "io"
    "os"
    "strings"
)

func main() {
    dec := xml.NewDecoder(os.Stdin)
    var stack []string // pilha de nomes de elementos
    for {
        tok, err := dec.Token()
        if err == io.EOF {
            break
        } else if err != nil {
            fmt.Fprintf(os.Stderr, "xmlselect: %v\n", err)
            os.Exit(1)
        }
        switch tok := tok.(type) {
        case xml.StartElement:
            stack = append(stack, tok.Name.Local) // faz um push
        case xml.EndElement:
            stack = stack[:len(stack)-1] // faz um pop
        case xml.CharData:
            if containsAll(stack, os.Args[1:]) {
                fmt.Printf("%s: %s\n", strings.Join(stack, " "), tok)
            }
        }
    }
}
```

```
// containsAll informa se x contém os elementos de y, na ordem.
func containsAll(x, y []string) bool {
    for len(y) <= len(x) {
        if len(y) == 0 {
            return true
        }
        if x[0] == y[0] {
            y = y[1:]
        }
        x = x[1:]
    }
    return false
}
```

Sempre que o loop em **main** encontra um **StartElement**, ele faz o push (insere) do nome do elemento em uma pilha e, para cada **EndElement**, faz um pop (remove) do nome da pilha. A API garante que haverá uma correspondência apropriada para a sequência de tokens **StartElement** e **EndElement**, mesmo em documentos malformados. **Comments** são ignorados. Quando **xmlselect** encontra um **CharData**, o texto é exibido somente se a pilha contiver todos os elementos nomeados pelos argumentos de linha de comando, na ordem.

O comando a seguir exibe o texto de qualquer elemento **h2** que apareça abaixo de dois níveis de elementos **div**. Sua entrada é a especificação XML que é, ela própria, um documento XML.

```
$ go build gopl.io/ch1/fetch
$ ./fetch http://www.w3.org/TR/2006/REC-xml11-20060816 |
  ./xmlselect div div h2
html body div div h2: 1 Introduction
html body div div h2: 2 Documents
html body div div h2: 3 Logical Structures
html body div div h2: 4 Physical Structures
html body div div h2: 5 Conformance
html body div div h2: 6 Notation
html body div div h2: A References
html body div div h2: B Definitions for Character Normalization
...
```

Exercício 7.17: Estenda **xmlselect** para que os elementos possam ser selecionados não só pelo nome, mas por seus atributos também, no estilo

de CSS, de modo que, por exemplo, um elemento como `<div id="page" class="wide">` possa ser selecionado por um `id` ou `class` correspondente assim como pelo seu nome.

Exercício 7.18: Usando a API do decodificador baseado em token, escreva um programa que leia um documento XML qualquer e construa uma árvore de nós genéricos que o represente. Os nós são de dois tipos: nós **CharData** representam strings de texto e nós **Element** representam elementos nomeados e seus atributos. Cada nó de elemento tem uma fatia de nós filhos.

Talvez você considere as declarações a seguir úteis.

```
import "encoding/xml"

type Node interface{} // CharData ou *Element
type CharData string
type Element struct {
    Type      xml.Name
    Attr      []xml.Attr
    Children  []Node
}
```

7.15 Alguns conselhos

Ao projetar um novo pacote, programadores iniciantes em Go muitas vezes começam criando um conjunto de interfaces e, somente mais tarde, definem os tipos concretos que as satisfazem. Essa abordagem resulta em muitas interfaces, cada qual com apenas uma única implementação. Não faça isso. Essas interfaces são abstrações desnecessárias e têm um custo em tempo de execução. Você pode restringir quais métodos de um tipo ou campos de uma estrutura são visíveis fora de um pacote usando o sistema de exportação (seção 6.6). Interfaces são necessárias somente quando houver dois ou mais tipos concretos com que devemos lidar de maneira uniforme.

Temos uma exceção a essa regra quando uma interface é satisfeita por um único tipo concreto, mas esse tipo não pode estar no mesmo pacote que a

interface por causa de suas dependências. Nesse caso, uma interface é uma boa maneira de desacoplar dois pacotes.

Como as interfaces são usadas em Go somente quando são satisfeitas por dois ou mais tipos, elas necessariamente abstraem os detalhes de qualquer implementação em particular. O resultado são interfaces menores, com métodos mais simples e em menor quantidade – geralmente apenas um, como no caso de `io.Writer` ou de `fmt.Stringer`. Interfaces menores são mais fáceis de satisfazer quando novos tipos surgem. Uma boa regra geral para o design de interfaces é *pedir apenas o que é necessário*.

Com isso, concluímos nosso tour pelos métodos e interfaces. Go tem um excelente suporte para o estilo de programação orientado a objetos, mas isso não quer dizer que você deva usá-lo exclusivamente. Nem tudo precisa ser um objeto; funções independentes têm seu lugar, assim como tipos de dados não encapsulados. Observe que, em conjunto, os exemplos dos cinco primeiros capítulos deste livro não chamam mais que duas dúzias de métodos, como `input.Scan`, em comparação a chamadas comuns de função como `fmt.Printf`.

8

Gorrotinas e canais

Programação concorrente – a expressão de um programa como uma composição de várias atividades autônomas – nunca foi tão importante quanto atualmente. Servidores web tratam requisições para milhares de clientes simultaneamente. Aplicativos para tablets e celulares apresentam animações na interface de usuário ao mesmo tempo que fazem cálculos e requisições de rede em background. Até mesmo problemas tradicionais em lote (batch) – ler alguns dados, processar, escrever uma saída – usam concorrência para ocultar a latência de operações de E/S e explorar os muitos processadores de um computador moderno, que todo ano crescem em número, mas não em velocidade.

Go possibilita dois estilos de programação concorrente. Este capítulo apresenta as gorrotinas e os canais, que tratam CSP (Communicating Sequential Processes, ou Processos Sequenciais Comunicantes): um modelo de concorrência em que valores são passados entre atividades independentes (gorrotinas), mas as variáveis são, em sua maior parte, confinadas em uma única atividade. O capítulo 9 discute alguns aspectos do modelo mais tradicional de *multithreading com memória compartilhada*, que será familiar se você já usou threads em outras linguagens populares. O capítulo 9 também destaca alguns perigos e armadilhas importantes da programação concorrente que não detalharemos neste capítulo.

Apesar do suporte de Go para concorrência ser um de seus principais pontos fortes, pensar em programas concorrentes é inerentemente mais difícil do que pensar em programas sequenciais, e a intuição adquirida com a programação sequencial às vezes pode nos deixar desorientados. Se essa é a primeira vez que você entra em contato com concorrência,

recomendamos investir um pouco de tempo extra analisando os exemplos destes dois capítulos.

8.1 Gorrotinas

Em Go, cada atividade que executa de forma concorrente é chamada de *gorrotina*. Considere um programa que tem duas funções: uma que faz um processamento e outra que escreve uma saída, e suponha que uma função não chame a outra. Um programa sequencial pode chamar uma função e depois chamar a outra, mas em um programa *concorrente* com duas ou mais gorrotinas, chamadas às *duas* funções podem estar ativas ao mesmo tempo. Veremos um programa desse tipo em breve.

Se você já usou threads do sistema operacional ou threads em outras linguagens, poderá supor, por enquanto, que uma gorrotina é semelhante a uma thread, e você será capaz de escrever programas corretos. As diferenças entre threads e gorrotinas são essencialmente quantitativas, e não qualitativas, e serão descritas na seção 9.8.

Quando um programa inicia, sua única gorrotina é aquela que chama a função `main`, portanto, nós a chamamos de *gorrotina principal* (main goroutine). Novas gorrotinas são criadas com a instrução `go`. Do ponto de vista sintático, uma instrução `go` é uma chamada comum de função ou de método prefixada pela palavra reservada `go`. Uma instrução `go` faz a função ser chamada em uma gorrotina recém-criada. A instrução `go` propriamente dita termina de imediato:

```
f()    // chama f(); espera que ela retorne
go f() // cria uma nova gorrotina que chama f(); não espera
```

No exemplo a seguir, a gorrotina principal calcula o quadragésimo quinto número de Fibonacci. Como usa o terrivelmente ineficiente algoritmo recursivo, o programa executa por um tempo considerável, durante o qual queremos oferecer ao usuário uma indicação visual de que o programa continua executando, exibindo um “spinner” textual animado.

gopl.io/ch8/spinner

```
func main() {
```



```

    go spinner(100 * time.Millisecond)
    const n = 45
    fibN := fib(n) // lento
    fmt.Printf("\rFibonacci(%d) = %d\n", n, fibN)
}

func spinner(delay time.Duration) {
    for {
        for _, r := range `-\|/` {
            fmt.Printf("\r%c", r)
            time.Sleep(delay)
        }
    }
}

func fib(x int) int {
    if x < 2 {
        return x
    }
    return fib(x-1) + fib(x-2)
}

```

Após vários segundos de animação, a chamada a **fib(45)** retorna, e a função **main** exibe seu resultado:

```
Fibonacci(45) = 1134903170
```

A função **main** então retorna. Quando isso acontece, todas as gorrotinas são abruptamente encerradas, e o programa termina. Exceto por retornar de **main** ou sair do programa, não há nenhuma maneira de uma gorrotina interromper outra por programação, mas, como veremos mais adiante, há modos de se comunicar com uma gorrotina para pedir que ela se interrompa.

Observe que o programa é expresso como a composição de duas atividades autônomas: girar o spinner e calcular o número de Fibonacci. Cada atividade é escrita como uma função separada, mas ambas fazem progresso concorrentemente.

8.2 Exemplo: Servidor de relógio concorrente

A rede é um domínio natural para usar concorrência, pois os servidores

em geral tratam muitas conexões de seus clientes ao mesmo tempo, nas quais cada cliente é essencialmente independente dos demais. Nesta seção, apresentaremos o pacote **net**, que oferece os componentes para criar programas clientes e servidores em rede; eles se comunicam por meio de TCP, UDP ou sockets de domínio Unix. O pacote **net/http**, que usamos desde o capítulo 1, foi desenvolvido com base nas funções do pacote **net**.

Nosso primeiro exemplo é um servidor de relógio sequencial que escreve o horário atual no cliente uma vez por segundo.

gopl.io/ch8/clock1

```
// Clock1 é um servidor TCP que escreve o horário periodicamente.
package main

import (
    "io"
    "log"
    "net"
    "time"
)

func main() {
    listener, err := net.Listen("tcp", "localhost:8000")
    if err != nil {
        log.Fatal(err)
    }
    for {
        conn, err := listener.Accept()
        if err != nil {
            log.Print(err) // por exemplo, conexão abortada
            continue
        }
        handleConn(conn) // trata uma conexão de cada vez
    }
}

func handleConn(c net.Conn) {
    defer c.Close()
    for {
        _, err := io.WriteString(c, time.Now().Format("15:04:05\n"))
        if err != nil {
            return // por exemplo, cliente desconectou
        }
    }
}
```

```
    }  
    time.Sleep(1 * time.Second)  
  }  
}
```

A função **Listen** cria um **net.Listener**: um objeto que ouve conexões de entrada em uma porta de rede, nesse caso, na porta TCP **localhost:8000**. O método **Accept** do listener fica bloqueado até que uma requisição de entrada para conexão seja feita e, então, devolve um objeto **net.Conn** que representa a conexão.

A função **handleConn** trata uma conexão completa de cliente. Em um loop, ela escreve o horário atual, **time.Now()**, no cliente. Como **net.Conn** satisfaz a interface **io.Writer**, podemos escrever diretamente nele. O loop termina quando a escrita falha, em especial porque o cliente desconectou; nesse instante, **handleConn** encerra seu lado da conexão usando uma chamada adiada para **Close** e volta a esperar outra solicitação de conexão.

O método **time.Time.Format** oferece uma maneira de formatar informações de data e hora por meio de exemplo. Seu argumento é um template que indica como formatar um horário de referência, especificamente **Mon Jan 2 03:04:05PM 2006 UTC-0700**. O horário de referência tem oito componentes (dia da semana, mês, dia do mês, e assim por diante). Qualquer coleção deles pode aparecer na string de **Format** em qualquer ordem e em formatos variados; os componentes selecionados da data e da hora serão apresentados nos formatos selecionados. Nesse caso, usamos apenas a hora, o minuto e o segundo do horário. O pacote **time** define templates para muitos formatos padrões de horário, como **time.RFC1123**. O mesmo sistema é usado ao inverso quando o parse de um horário é feito com **time.Parse**.

Para se conectar ao servidor, precisaremos de um programa cliente como **nc** (“netcat”), que é um utilitário padrão para manipulação de conexões de rede:

```
$ go build gopl.io/ch8/clock1  
$ ./clock1 &  
$ nc localhost 8000
```

```
13:58:54
13:58:55
13:58:56
13:58:57
^C
```

O cliente exibe o horário enviado pelo servidor a cada segundo até o interrompermos com Control-C, que em sistemas Unix é ecoado como **^C** pelo shell. Se o **nc** ou o **netcat** não estiver instalado em seu sistema, você poderá usar **telnet** ou a versão simples de **netcat** em Go, a seguir, que utiliza **net.Dial** para conectar-se a um servidor TCP:

gopl.io/ch8/netcat1

```
// Netcat1 é um cliente TCP somente para leitura.
package main

import (
    "io"
    "log"
    "net"
    "os"
)

func main() {
    conn, err := net.Dial("tcp", "localhost:8000")
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()
    mustCopy(os.Stdout, conn)
}

func mustCopy(dst io.Writer, src io.Reader) {
    if _, err := io.Copy(dst, src); err != nil {
        log.Fatal(err)
    }
}
```

Esse programa lê dados da conexão e escreve-os na saída-padrão até uma condição de fim de arquivo (end-of-file) ou um erro ocorrer. A função **mustCopy** é um utilitário que aparece em vários exemplos desta seção. Vamos executar dois clientes ao mesmo tempo em terminais diferentes, um mostrado à esquerda e o outro à direita:

```

$ go build gopl.io/ch8/netcat1
$ ./netcat1
13:58:54
13:58:55
13:58:56
^C

13:58:57
13:58:58
13:58:59
^C

```

```
$ killall clock1
```

O comando **killall** é um utilitário do Unix que mata todos os processos com o nome especificado.

O segundo cliente deve esperar até o primeiro terminar, pois o servidor é *sequencial*; ele só trata um cliente de cada vez. Apenas uma pequena modificação é necessária para deixar o servidor concorrente: acrescentar a palavra reservada **go** à chamada de **handleConn** faz cada chamada ser executada em sua própria gorrotina.

gopl.io/ch8/clock2

```

for {
    conn, err := listener.Accept()
    if err != nil {
        log.Print(err)    // por exemplo, conexão abortada
        continue
    }
    go handleConn(conn) // trata conexões de forma concorrente
}

```

Agora vários clientes podem receber o horário ao mesmo tempo:

```

$ go build gopl.io/ch8/clock2
$ ./clock2 &
$ go build gopl.io/ch8/netcat1
$ ./netcat1
14:02:54
14:02:55
14:02:56
14:02:57
14:02:58
14:02:59

$ ./netcat1
14:02:55
14:02:56
^C
$ ./netcat1

```

14:03:00

14:03:01

^C

14:03:00

14:03:01

14:03:02

^C

\$ killall clock2

Exercício 8.1: Modifique `clock2` para que aceite um número de porta e escreva um programa `clockwall` que atue como um cliente de vários servidores de relógio ao mesmo tempo, lendo os horários de cada um e exibindo os resultados em uma tabela, semelhante a uma parede com relógios que vemos em alguns escritórios. Se você tiver acesso a computadores distribuídos geograficamente, execute instâncias remotamente; caso contrário, execute instâncias locais em portas diferentes com fusos horários simulados.

\$ TZ=US/Eastern ./clock2 -port 8010 &

\$ TZ=Asia/Tokyo ./clock2 -port 8020 &

\$ TZ=Europe/London ./clock2 -port 8030 &

\$ clockwall NewYork=localhost:8010 Tokyo=localhost:8020 London=localhost:8030

Exercício 8.2: Implemente um servidor de FTP (File Transfer Protocol, ou Protocolo de Transferência de Arquivos) concorrente. O servidor deve interpretar os comandos de cada cliente, como `cd` para mudança de diretório, `ls` para listar um diretório, `get` para enviar o conteúdo de um arquivo e `close` para encerrar a conexão. Você pode usar o comando `ftp` padrão como cliente ou escrever seu próprio cliente.

8.3 Exemplo: Servidor de eco concorrente

O servidor de relógio usou uma gorrotina por conexão. Nesta seção, criaremos um servidor de eco que utiliza várias gorrotinas por conexão. A maioria dos servidores de eco simplesmente escreve o que leem; isso pode ser feito com esta versão trivial de `handleConn`:

```
func handleConn(c net.Conn) {
    io.Copy(c, c) // NOTA: ignorando erros
    c.Close()
}
```

Um servidor de eco mais interessante pode simular as reverberações de

um eco de verdade, com a resposta enfática no início ("HELLO!"), em seguida moderada ("Hello!") após um intervalo de tempo e, então, murmurando ("hello!") antes de silenciar, como nesta versão de `handleConn`:

gopl.io/ch8/reverb1

```
func echo(c net.Conn, shout string, delay time.Duration) {
    fmt.Fprintln(c, "\t", strings.ToUpper(shout))
    time.Sleep(delay)
    fmt.Fprintln(c, "\t", shout)
    time.Sleep(delay)
    fmt.Fprintln(c, "\t", strings.ToLower(shout))
}

func handleConn(c net.Conn) {
    input := bufio.NewScanner(c)
    for input.Scan() {
        echo(c, input.Text(), 1*time.Second)
    }
    // NOTA: ignorando erros em potencial de input.Err()
    c.Close()
}
```

Precisaremos fazer um upgrade em nosso programa cliente para que ele envie a entrada do terminal para o servidor, ao mesmo tempo que também copia a resposta do servidor na saída, o que representa outra oportunidade para usar concorrência:

gopl.io/ch8/netcat2

```
func main() {
    conn, err := net.Dial("tcp", "localhost:8000")
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()
    go mustCopy(os.Stdout, conn)
    mustCopy(conn, os.Stdin)
}
```

Enquanto a gorrotina principal lê a entrada-padrão e a envia ao servidor, uma segunda gorrotina lê e exibe a resposta do servidor. Quando a gorrotina principal encontra o final da entrada, por exemplo, depois que o

usuário digita Control-D (^D) no terminal (ou o equivalente Control-Z no Microsoft Windows), o programa para, mesmo que a outra gorrotina continue com trabalho pendente. (Veremos como fazer o programa esperar que ambos os lados terminem depois que apresentarmos os canais na seção 8.4.1.)

Na sessão a seguir, a entrada do cliente está alinhada à esquerda, e as respostas do servidor estão indentadas. O cliente grita três vezes para o servidor de eco:

```
$ go build gopl.io/ch8/reverb1
$ ./reverb1 &
$ go build gopl.io/ch8/netcat2
$ ./netcat2
Hello?
    HELLO?
    Hello?
    hello?
Is there anybody there?
    IS THERE ANYBODY THERE?
Yooo-hooo!
    Is there anybody there?
    is there anybody there?
    Y000-H000!
    Yooo-hooo!
    yooo-hooo!
^D
$ killall reverb1
```

Observe que o terceiro grito do cliente não é tratado até que o segundo grito tenha se desvanecido, o que não é muito realista. Um eco de verdade seria constituído da *composição* dos três gritos independentes. Para simular isso, precisaremos de mais gorrotinas. Novamente, tudo que precisamos fazer é acrescentar a palavra reservada **go**, desta vez, na chamada a **echo**:

[gopl.io/ch8/reverb2](#)

```
func handleConn(c net.Conn) {
    input := bufio.NewScanner(c)
    for input.Scan() {
        go echo(c, input.Text(), 1*time.Second)
```



```

    }
    // NOTA: ignorando erros em potencial de input.Err()
    c.Close()
}

```

Os argumentos da função iniciada por **go** são avaliados quando a própria instrução **go** é executada; desse modo, **input.Text()** é avaliado na gorrotina principal.

Agora os ecos são concorrentes e se sobrepõem no tempo:

```

$ go build gopl.io/ch8/reverb2
$ ./reverb2 &
$ ./netcat2
Is there anybody there?
  IS THERE ANYBODY THERE?
Yooo-hooo!
  Is there anybody there?
  Y000-H000!
  is there anybody there?
  Yooo-hooo!
  yooo-hooo!
^D
$ killall reverb2

```

Tudo o que foi necessário para fazer o servidor usar concorrência, não só para tratar conexões de vários clientes, mas até mesmo em uma única conexão, foi inserir as duas palavras reservadas **go**.

No entanto, ao adicionar essas palavras reservadas, foi preciso considerar cuidadosamente se é seguro chamar métodos de **net.Conn** de forma concorrente, o que não é verdade para a maioria dos tipos. Discutiremos o conceito fundamental de *segurança em concorrência* no próximo capítulo.

8.4 Canais

Se gorrotinas são as atividades de um programa Go concorrente, *canais* (channels) são as conexões entre elas. Um canal é um sistema de comunicação que permite a uma gorrotina enviar valores para outra gorrotina. Cada canal é um condutor de valores de um tipo particular, chamado de *tipo de elemento* do canal. O tipo de um canal cujos

elementos têm tipo `int` é escrito como `chan int`.

Para criar um canal, usamos a função embutida `make`:

```
ch := make(chan int) // ch é do tipo 'chan int'
```

Como no caso dos mapas, um canal é uma *referência* à estrutura de dados criada por `make`. Quando copiamos um canal ou passamos um como argumento de uma função, estamos copiando uma referência, portanto, quem chama e quem foi chamado referem-se à mesma estrutura de dados. Como ocorre com outros tipos referência, o valor zero de um canal é `nil`.

Dois canais de mesmo tipo podem ser comparados com `==`. A comparação é verdadeira se ambos forem referências à estrutura de dados do mesmo canal. Um canal também pode ser comparado a `nil`.

Um canal tem duas operações principais, *enviar* (send) e *receber* (receive) que, em conjunto, são conhecidas como *comunicações*. Uma instrução de envio transmite um valor de uma gorrotina, por meio do canal, a outra gorrotina, que executa uma expressão de recepção correspondente. Ambas as operações são escritas usando o operador `<-`. Em uma instrução de envio, `<-` separa o canal e os operandos contendo o valor. Em uma expressão de recepção, `<-` antecede o operando referente ao canal. Uma expressão de recepção cujo resultado não seja usado é uma instrução válida.

```
ch <- x // uma instrução de envio
```

```
x = <-ch // uma instrução de recepção em uma instrução de atribuição
```

```
<-ch // uma instrução de recepção; o resultado é descartado
```

Canais aceitam uma terceira operação, *fechar* (close), que define uma flag indicando que valores não serão mais enviados por esse canal; tentativas subsequentes de envio geram pânico. Operações de recepção em um canal fechado produzem os valores que foram enviados até que não restem mais valores; qualquer operação de recepção depois disso se completa imediatamente e produz o valor zero do tipo do elemento do canal.

Para fechar um canal, chamamos a função embutida `close`:

```
close(ch)
```

Um canal criado com uma chamada simples a `make` chama-se canal *sem*

buffer (unbuffered channel), mas **make** aceita um segundo argumento opcional: um inteiro que é a *capacidade* (capacity) do canal. Se a capacidade for diferente de zero, **make** cria um canal *com buffer* (buffered channel).

```
ch = make(chan int)    // canal sem buffer
ch = make(chan int, 0) // canal sem buffer
ch = make(chan int, 3) // canal com buffer de capacidade 3
```

Daremos uma olhada nos canais sem buffer antes e nos canais com buffer na seção 8.4.4.

8.4.1 Canais sem buffer

Uma operação de envio em um canal sem buffer bloqueia a gorrotina que envia, até outra gorrotina executar uma recepção correspondente no mesmo canal; nesse ponto, o valor é transmitido, e ambas as gorrotinas podem continuar. Por outro lado, se houver uma tentativa de operação de recepção antes, a gorrotina receptora é bloqueada até outra gorrotina executar um envio no mesmo canal.

A comunicação por meio de um canal sem buffer faz as gorrotinas que enviam e que recebem se *sincronizarem*. Por causa disso, canais sem buffer às vezes são chamados de canais *síncronos*. Quando um valor é enviado por meio de um canal sem buffer, a recepção do valor *acontece antes* de a gorrotina que envia acordar novamente.

Em discussões sobre concorrência, quando dizemos que *x acontece antes de y*, não queremos simplesmente dizer que *x* ocorre antes de *y* no tempo; queremos dizer que isso é garantido e que todos os seus efeitos anteriores, como atualizações em variáveis, serão concluídos e que você poderá contar com eles.

Quando *x* não acontece antes de *y* nem depois de *y*, dizemos que *x é concorrente com y*. Isso não quer dizer que *x* e *y* sejam necessariamente simultâneos; quer dizer apenas que não podemos supor nada sobre sua ordem. Como veremos no próximo capítulo, é necessário ordenar determinados eventos durante a execução do programa para evitar os

problemas que surgem quando duas gorrotinas acessam a mesma variável de forma concorrente.

O programa cliente da seção 8.3 copia dados de entrada para o servidor em sua gorrotina principal, portanto o programa cliente termina assim que o stream de entrada é fechado, mesmo que a gorrotina em background continue funcionando. Para fazer o programa esperar que a gorrotina em background termine antes de sair, usamos um canal para sincronizar as duas gorrotinas:

gopl.io/ch8/netcat3

```
func main() {
    conn, err := net.Dial("tcp", "localhost:8000")
    if err != nil {
        log.Fatal(err)
    }
    done := make(chan struct{})
    go func() {
        io.Copy(os.Stdout, conn) // NOTA: ignorando erros
        log.Println("done")
        done <- struct{}{} // sinaliza para a gorrotina principal
    }()
    mustCopy(conn, os.Stdin)
    conn.Close()
    <-done // espera a gorrotina em background terminar
}
```

Quando o usuário fecha o stream da entrada-padrão, **mustCopy** retorna, e a gorrotina principal chama **conn.Close()**, fechando ambas as metades da conexão de rede. Fechar a metade de escrita da conexão faz o servidor ver uma condição de fim de arquivo. Fechar a metade de leitura faz a chamada a **io.Copy** feita pela gorrotina em background devolver um erro de “leitura de conexão fechada”, motivo pelo qual removemos o logging de erro; o exercício 8.3 sugere uma solução melhor. (Observe que a instrução **go** chama uma função literal, que é uma construção comum.)

Antes de retornar, a gorrotina em background faz log de uma mensagem e então envia um valor pelo canal **done**. A gorrotina principal espera até receber esse valor antes de retornar. Como resultado, o programa sempre

loga a mensagem "**done**" antes de sair.

Mensagens enviadas pelos canais têm dois aspectos importantes. Cada mensagem tem um valor, mas, às vezes, o fato da comunicação e o momento em que ela ocorre são igualmente importantes. Chamamos as mensagens de *eventos* quando queremos enfatizar esse aspecto. Quando o evento não transporta informações adicionais, isto é, seu único propósito é a sincronização, enfatizaremos isso usando um canal cujo tipo de elemento é `struct{}`, embora seja comum usar um canal com `bool` ou com `int` para a mesma finalidade, pois `done <- 1` é mais conciso que `done <- struct{}{}`.

Exercício 8.3: Em `netcat3`, o valor da interface `conn` tem o tipo concreto `*net.TCPConn`, que representa uma conexão TCP. Uma conexão TCP é constituída de duas metades que podem ser fechadas de forma independente usando seus métodos `CloseRead` e `CloseWrite`. Modifique a gorrotina principal de `netcat3` para fechar somente a metade de escrita da conexão de modo que o programa continue a exibir os ecos finais do servidor `reverb1` mesmo depois de a entrada-padrão ter sido fechada. (Fazer isso para o servidor `reverb2` é mais difícil; veja o exercício 8.4.)

8.4.2 Pipelines

Canais podem ser usados para conectar gorrotinas de modo que a saída de uma seja a entrada de outra. Isso se chama *pipeline*. O programa a seguir é constituído de três gorrotinas conectadas por dois canais, como mostra o esquema da figura 8.1.

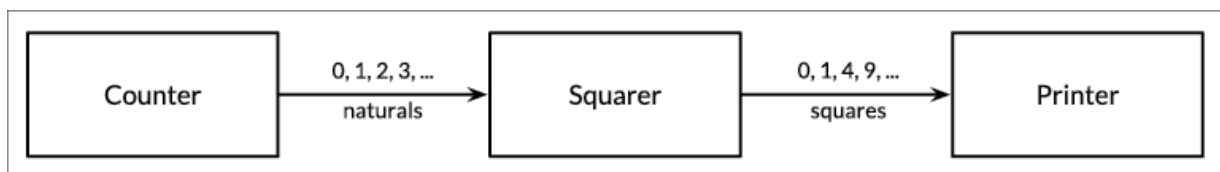


Figura 8.1 – Um pipeline de três estágios.

A primeira gorrotina, *counter*, gera os inteiros 0, 1, 2, ..., e os envia por um canal para a segunda gorrotina, *squarer*, que recebe cada valor, calcula seu quadrado e envia o resultado por outro canal para a terceira gorrotina,

printer, que recebe os valores ao quadrado e os exhibe. Por questões de clareza, nesse exemplo, escolhemos intencionalmente funções bem simples, embora, é claro, elas sejam triviais demais do ponto de vista de processamento para merecerem suas próprias gorrotinas em um programa realista.

gopl.io/ch8/pipeline1

```
func main() {
    naturals := make(chan int)
    squares := make(chan int)

    // Counter
    go func() {
        for x := 0; ; x++ {
            naturals <- x
        }
    }()

    // Squarer
    go func() {
        for {
            x := <-naturals
            squares <- x * x
        }
    }()

    // Printer (na gorrotina principal)
    for {
        fmt.Println(<-squares)
    }
}
```

Como seria de esperar, o programa exhibe a série infinita de quadrados, 0, 1, 4, 9, e assim sucessivamente. Pipelines como esse podem ser encontrados em programas servidores que executam por muito tempo, em que canais são usados para comunicação entre gorrotinas contendo loops infinitos durante todo o tempo de vida do programa. Mas e se quiséssemos enviar apenas um número finito de valores pelo pipeline?

Se quem faz o envio sabe que não haverá outros valores a serem enviados em um canal, é conveniente comunicar esse fato às gorrotinas receptoras para que elas possam parar de esperar. Isso é feito *fechando* o canal por

meio da função embutida **close**:

```
close(naturals)
```

Depois que um canal é fechado, qualquer outra operação de envio por ele gerará pânico. Após o canal fechado ter sido *drenado* (drained), isto é, depois que o último elemento enviado for recebido, todas as operações subsequentes de recepção continuarão sem bloqueio, mas produzirão um valor zero. Fechar o canal **naturals** anterior faria o loop de squarer executar rapidamente, pois ele receberia um fluxo interminável de valores zero, que seriam enviados a printer.

Não há meios de testar diretamente se um canal foi fechado, mas há uma variante da operação de recepção que gera dois resultados: o elemento recebido do canal, mais um valor booleano, chamado de **ok** por convenção, que é **true** para uma recepção bem-sucedida e **false** para uma recepção em um canal fechado e drenado. Usando esse recurso, podemos modificar o loop de squarer para parar quando o canal **naturals** estiver drenado e fechar o canal **squares**, por sua vez.

```
// Squarer
go func() {
    for {
        x, ok := <-naturals
        if !ok {
            break // o canal foi fechado e drenado
        }
        squares <- x * x
    }
    close(squares)
}()
```

Como a sintaxe anterior é desajeitada e esse padrão é comum, a linguagem permite usar um loop **range** para iterar pelos canais também. Essa é uma sintaxe mais conveniente para receber todos os valores enviados em um canal e encerrar o loop após o último valor.

No pipeline a seguir, quando a gorrotina counter encerra seu loop após cem elementos, ela fecha o canal **naturals** fazendo squarer terminar seu loop e fechar o canal **squares**. (Em um programa mais complexo, pode

fazer sentido que as funções `counter` e `squarer` chamem **close** de forma adiada já no início.) Por fim, a gorrotina principal encerra seu loop, e o programa termina.

gopl.io/ch8/pipeline2

```
func main() {
    naturals := make(chan int)
    squares := make(chan int)

    // Counter
    go func() {
        for x := 0; x < 100; x++ {
            naturals <- x
        }
        close(naturals)
    }()

    // Squarer
    go func() {
        for x := range naturals {
            squares <- x * x
        }
        close(squares)
    }()

    // Printer (na gorrotina principal)
    for x := range squares {
        fmt.Println(x)
    }
}
```

Não é preciso fechar todos os canais quando acabar de usá-los. É necessário fechar um canal somente quando for importante dizer às gorrotinas receptoras que todos os dados foram enviados. Um canal que o coletor de lixo determine que seja inacessível terá seus recursos liberados, independentemente de ter sido ou não fechado. (Não confunda isso com a operação de fechamento para arquivos abertos. É importante chamar o método **Close** em todos os arquivos quando você acabar de usá-los.)

Tentar fechar um canal já fechado gera pânico, assim como fechar um canal `nil`. Fechar canais tem outro uso como um sistema de broadcast, que discutiremos na seção 8.9.

8.4.3 Canais unidirecionais

À medida que os programas crescem, é natural dividir funções grandes em partes menores. Nosso exemplo anterior usou três gorrotinas que se comunicavam por meio de dois canais, que eram variáveis locais de `main`. O programa naturalmente se divide em três funções:

```
func counter(out chan int)
func squarer(out, in chan int)
func printer(in chan int)
```

A função **squarer**, que fica no meio do pipeline, aceita dois parâmetros: o canal de entrada e o canal de saída. Ambos são do mesmo tipo, mas o uso a que se destinam são opostos: **in** é somente para recepção e **out** é somente para envio. Os nomes **in** e **out** traduzem essa intenção, mas, apesar disso, nada impede **squarer** de enviar para **in** ou receber de **out**.

Essa organização é característica. Quando um canal é fornecido como um parâmetro de função, quase sempre é com a intenção de que ele seja usado exclusivamente para enviar ou exclusivamente para receber.

Para documentar esse propósito e evitar um uso indevido, o sistema de tipos de Go oferece tipos de canais *unidirecionais* que expõem apenas uma das operações, de envio ou de recepção. O tipo `chan<- int`, um canal de **int** *somente para envio* (send-only), permite enviar, mas não receber. Por outro lado, o tipo `<-chan int`, um canal de **int** *somente para recepção* (receive-only), permite receber, mas não enviar. (A posição da seta `<-` em relação à palavra reservada **chan** é um mnemônico.) Violações dessa regra são identificadas em tempo de compilação.

Como a operação **close** garante que nenhum outro envio ocorrerá em um canal, apenas a gorrotina que envia pode chamá-la e, por esse motivo, é um erro de tempo de compilação tentar fechar um canal somente de recepção.

Eis novamente o pipeline que calcula os quadrados, desta vez com tipos unidirecionais de canais:

gopl.io/ch8/pipeline3

```
func counter(out chan<- int) {
```

```

    for x := 0; x < 100; x++ {
        out <- x
    }
    close(out)
}

func squarer(out chan<- int, in <-chan int) {
    for v := range in {
        out <- v * v
    }
    close(out)
}

func printer(in <-chan int) {
    for v := range in {
        fmt.Println(v)
    }
}

func main() {
    naturals := make(chan int)
    squares := make(chan int)
    go counter(naturals)
    go squarer(squares, naturals)
    printer(squares)
}

```

A chamada a `counter(naturals)` implicitamente converte `naturals`, um valor do tipo `chan int`, para o tipo do parâmetro, `chan<- int`. A chamada a `printer(squares)` faz uma conversão implícita semelhante para `<-chan int`. Conversões de tipos bidirecionais para unidirecionais de canais são permitidas em qualquer atribuição. Porém, não há caminho de volta: depois que você tiver um valor para um tipo unidirecional como `chan<- int`, não há como obter um valor do tipo `chan int` a partir dele que se refira à mesma estrutura de dados de canal.

8.4.4 Canais com buffer

Um canal com buffer tem uma fila de elementos. O tamanho máximo da fila é determinado quando o canal é criado, por meio do argumento de capacidade em `make`. A instrução a seguir cria um canal com buffer capaz de armazenar três valores do tipo `string`. A figura 8.2 mostra uma

representação gráfica de `ch` e do canal ao qual ele se refere.

```
ch = make(chan string, 3)
```

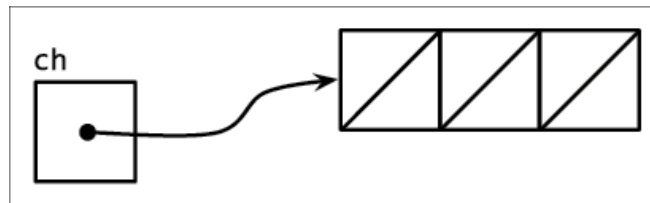


Figura 8.2 – Um canal com buffer vazio.

Uma operação de envio em um canal com buffer insere um elemento no final da fila, e uma operação de recepção remove um elemento do início. Se o canal estiver cheio, a operação de envio bloqueia sua gorrotina até que um espaço seja liberado pela recepção em outra gorrotina. Por outro lado, se o canal estiver vazio, uma operação de recepção permanece bloqueada até um valor ser enviado por outra gorrotina.

Podemos enviar até três valores neste canal sem que a gorrotina fique bloqueada:

```
ch <- "A"  
ch <- "B"  
ch <- "C"
```

A essa altura, o canal está cheio (Figura 8.3) e uma quarta instrução de envio seria bloqueada.

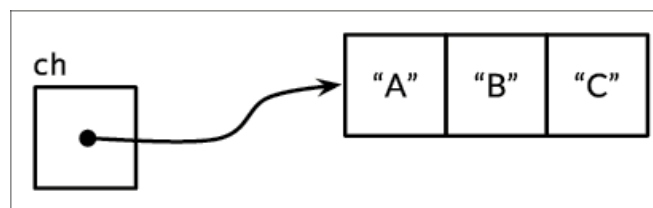


Figura 8.3 – Um canal com buffer cheio.

Se recebermos um valor:

```
fmt.Println(<-ch) // "A"
```

o canal não estará nem cheio nem vazio (Figura 8.4). Assim, tanto uma operação de envio quanto uma operação de recepção poderiam prosseguir sem bloqueio. Dessa maneira, o buffer do canal desacopla as gorrotinas de envio e de recepção.

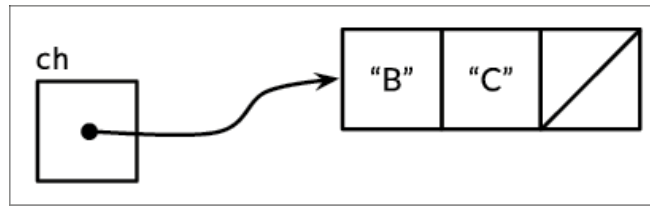


Figura 8.4 – Um canal com buffer parcialmente cheio.

Na improvável eventualidade de um programa precisar saber a capacidade do buffer do canal, essa informação pode ser obtida chamando a função embutida **cap**:

```
fmt.Println(cap(ch)) // "3"
```

Quando aplicada a um canal, a função embutida **len** devolve o número de elementos no buffer no momento. Como em um programa concorrente essa informação provavelmente estará desatualizada assim que for obtida, sua utilidade é limitada, mas poderia ser supostamente útil durante um diagnóstico de falhas ou uma otimização de desempenho.

```
fmt.Println(len(ch)) // "2"
```

Após outras duas operações de recepção, o canal estará vazio outra vez, e uma quarta operação ficaria bloqueada.

```
fmt.Println(<-ch) // "B"  
fmt.Println(<-ch) // "C"
```

Nesse exemplo, todas as operações de envio e de recepção foram feitas pela mesma gorrotina, mas, em programas reais, elas geralmente são executadas por gorrotinas diferentes. Às vezes, os iniciantes são tentados a usar canais com buffer em uma única gorrotina como uma fila, seduzidos pela sintaxe agradavelmente simples, mas isso é um erro. Canais estão profundamente associados a escalonamento de gorrotinas e, sem outra gorrotina recebendo do canal, quem envia – e talvez o programa todo – corre o risco de ficar bloqueado para sempre. Se tudo de que você precisa é de uma fila simples, crie uma usando uma fatia.

O exemplo a seguir mostra uma aplicação de um canal com buffer. Ele faz requisições paralelas a três *espelhos* (mirrors), isto é, servidores equivalentes, porém geograficamente distribuídos. Suas respostas são enviadas por meio de um canal com buffer; então apenas a primeira

resposta é recebida e devolvida, que é a mais rápida a chegar. Assim `mirroredQuery` devolve um resultado mesmo antes de os dois servidores mais lentos terem respondido. (A propósito, é bem comum que várias gorrotinas enviem valores ao mesmo canal de forma concorrente, como nesse exemplo, ou recebam do mesmo canal.)

```
func mirroredQuery() string {
    responses := make(chan string, 3)
    go func() { responses <- request("asia.gopl.io") }()
    go func() { responses <- request("europe.gopl.io") }()
    go func() { responses <- request("americas.gopl.io") }()
    return <-responses // devolve a resposta mais rápida
}

func request(hostname string) (response string) { /* ... */ }
```

Se tivéssemos usado um canal sem buffer, as duas gorrotinas mais lentas ficariam travadas tentando enviar suas respostas em um canal do qual nenhuma gorrotina faria a recepção. Essa situação, chamada de *vazamento de gorrotina* (goroutine leak), é um bug. Diferente de variáveis inacessíveis, gorrotinas vazadas não são coletadas automaticamente, portanto, é importante garantir que elas encerrem a si mesmas quando não são mais necessárias.

A escolha entre canais com e sem buffer e a escolha da capacidade de um canal com buffer podem afetar o fato de um programa estar correto. Canais sem buffer oferecem mais garantias de sincronização, pois toda operação de envio é sincronizada com sua recepção correspondente; em canais com buffer, essas operações são desacopladas. Além disso, quando conhecemos o limite máximo para a quantidade de valores que será enviada em um canal, não é incomum criar um canal com buffer com esse tamanho e realizar todos os envios antes de o primeiro valor ser recebido. Falha em alocar capacidade suficiente de buffer fará o programa entrar em deadlock.

O uso de canais com buffer também pode afetar o desempenho do programa. Suponha que haja três confeitheiros em uma confeitaria, um assando bolos, outro cobrindo com glacê e outro decorando cada bolo antes de passá-lo para o próximo confeitheiro na linha de produção. Em

uma cozinha com pouco espaço, cada confeitiro que terminar um bolo deve esperar que o próximo confeitiro esteja pronto para aceitá-lo; essa organização é análoga à comunicação por meio de um canal sem buffer.

Se houver espaço para um bolo entre cada confeitiro, um confeitiro pode colocar um bolo pronto ali e começar a trabalhar imediatamente no próximo; isso é análogo a um canal com buffer de capacidade um. Desde que os confeitiros trabalhem aproximadamente com a mesma velocidade média, a maioria dessas passagens ocorre de forma rápida, atenuando diferenças momentâneas em suas respectivas velocidades. Mais espaço entre os confeitiros – buffers maiores – pode atenuar maiores variações temporárias em suas velocidades, sem interromper a linha de produção, como ocorre quando um confeitiro faz um rápido intervalo e depois se apressa para tirar o atraso.

Por outro lado, se um estágio inicial da linha de produção for consistentemente mais rápido que o estágio seguinte, o buffer entre eles passará a maior parte do tempo cheio. Em comparação, se o estágio final for mais rápido, o buffer geralmente estará vazio. Um buffer não oferece nenhuma vantagem nesse caso.

A metáfora da linha de produção é útil para canais e gorrotinas. Por exemplo, se o segundo estágio for mais complexo, um único confeitiro talvez não seja capaz de dar conta do fornecimento do primeiro confeitiro nem atender à demanda do terceiro. Para resolver o problema, poderíamos contratar outro confeitiro para ajudar o segundo, executando a mesma tarefa, mas trabalhando independentemente. Isso é análogo a criar outra gorrotina que se comunicará pelos mesmos canais.

Não temos espaço para mostrá-lo aqui, mas o pacote `gopl.io/ch8/cake` simula essa confeitaria, com diversos parâmetros que você pode variar. Ele inclui benchmarks (seção 11.4) para alguns dos cenários descritos anteriormente.

8.5 Looping em paralelo

Nesta seção, exploraremos alguns padrões comuns de concorrência para

executar todas as iterações de um loop em paralelo. Consideraremos o problema de produzir imagens em miniatura (thumbnails) a partir de um conjunto de imagens de tamanho maior. O pacote `gopl.io/ch8/thumbnail` oferece uma função `ImageFile` capaz de reduzir uma única imagem. Não mostraremos sua implementação, mas você pode fazer o download a partir de `gopl.io`.

`gopl.io/ch8/thumbnail`

```
package thumbnail
// ImageFile lê uma imagem de infile e grava
// uma versão em miniatura dela no mesmo diretório.
// Ela devolve o nome do arquivo gerado, por exemplo, "foo.thumb.jpg".
func ImageFile(infile string) (string, error)
```

O programa a seguir percorre uma lista de nomes de arquivos de imagem em um loop e gera uma miniatura de cada imagem:

`gopl.io/ch8/thumbnail`

```
// makeThumbnails cria miniaturas dos arquivos especificados.
func makeThumbnails(filenames []string) {
    for _, f := range filenames {
        if _, err := thumbnail.ImageFile(f); err != nil {
            log.Println(err)
        }
    }
}
```

É claro que a ordem em que processamos os arquivos não importa, pois cada operação de redução é independente de todas as demais. Problemas como esse, que consistem de subproblemas totalmente independentes uns dos outros, são descritos como *embaraçosamente paralelos* (embarrassingly parallel). Esse tipo de problema é o mais simples para implementar de forma concorrente e desfruta de um desempenho que escala linearmente com o volume de paralelismo.

Vamos executar todas essas operações em paralelo, ocultando assim a latência da E/S de arquivo e usando várias CPUs para o processamento de escala de imagens. Nossa primeira tentativa de uma versão concorrente simplesmente adiciona a palavra reservada `go`. Vamos ignorar erros por

enquanto e cuidar deles depois.

```
// NOTA: incorreto!
func makeThumbnails2(filenames []string) {
    for _, f := range filenames {
        go thumbnail.ImageFile(f) // NOTA: ignorando erros
    }
}
```

Essa versão é realmente bem rápida – rápida demais, na verdade, pois demora menos que a original, mesmo quando a fatia com os nomes dos arquivos contém apenas um único elemento. Se não há paralelismo, como a versão concorrente pode executar mais rápido? A resposta é que **makeThumbnails** retorna antes de acabar de fazer o que deve. Ela inicia todas as gorrotinas, uma por nome de arquivo, mas não espera que elas terminem.

Não há nenhuma maneira direta de esperar uma gorrotina terminar, mas podemos alterar a gorrotina interna para que informe à gorrotina mais externa que terminou enviando um evento em um canal compartilhado. Como sabemos que há exatamente **len(filenames)** gorrotinas internas, a gorrotina externa só precisa contar essa quantidade de eventos antes de retornar:

```
// makeThumbnails3 cria miniaturas dos arquivos especificados, em paralelo.
func makeThumbnails3(filenames []string) {
    ch := make(chan struct{})
    for _, f := range filenames {
        go func(f string) {
            thumbnail.ImageFile(f) // NOTA: ignorando erros
            ch <- struct{}{}
        }(f)
    }

    // Espera as gorrotinas terminarem.
    for range filenames {
        <-ch
    }
}
```

Observe que passamos o valor de **f** como um argumento explícito à função literal em vez de usar a declaração de **f** do loop **for** que a engloba:


```

for _, f := range filenames {
    go func() {
        thumbnail.ImageFile(f) // NOTA: incorreto!
        // ...
    }()
}

```

Lembre-se do problema de captura de variável de loop em uma função anônima, descrito na seção 5.6.1. No código anterior, a variável única **f** é compartilhada por todos os valores de funções anônimas e é atualizada por sucessivas iterações do loop. Quando as novas gorrotinas começam a executar a função literal, o loop **for** pode ter atualizado **f** e iniciado outra iteração ou (provavelmente) ter terminado, portanto, quando essas gorrotinas lerem o valor de **f**, todas elas verão que ela tem o valor do último elemento da fatia. Ao acrescentar um parâmetro explícito, garantiremos que será capturado o valor de **f** do instante em que a instrução **go** é executada.

E se quiséssemos devolver valores de cada gorrotina de trabalho (worker) para a gorrotina principal? Se a chamada a **thumbnail.ImageFile** falhar em criar um arquivo, ela devolve um erro. A próxima versão de **makeThumbnails** devolve o primeiro erro que ela receber de qualquer uma das operações de escala:

```

// makeThumbnails4 cria miniaturas dos arquivos especificados, em paralelo.
// Ela devolve um erro se algum passo falhar.
func makeThumbnails4(filenames []string) error {
    errors := make(chan error)

    for _, f := range filenames {
        go func(f string) {
            _, err := thumbnail.ImageFile(f)
            errors <- err
        }(f)
    }

    for range filenames {
        if err := <-errors; err != nil {
            return err // NOTA: incorreto: vazamento de gorrotina!
        }
    }

    return nil
}

```

```
}
```

Essa função tem um bug sutil. Quando encontra o primeiro erro diferente de nil, ela devolve o erro a quem chamou, fazendo com que nenhuma gorrotina drene o canal **errors**. As demais gorrotinas de trabalho ficarão bloqueadas para sempre quando tentarem enviar um valor através desse canal, e jamais terminarão. Essa situação – um vazamento de gorrotina (seção 8.4.4) – pode fazer todo o programa ficar bloqueado ou sem memória.

A solução mais simples é usar um canal com buffer, com capacidade suficiente, de modo que nenhuma gorrotina de trabalho fique bloqueada quando enviar uma mensagem. (Uma solução alternativa é criar outra gorrotina para drenar o canal enquanto a gorrotina principal devolve o primeiro erro sem demora.)

A próxima versão de **makeThumbnails** usa um canal com buffer para devolver os nomes dos arquivos de imagem gerados, juntamente com qualquer erro.

```
// makeThumbnails5 cria miniaturas dos arquivos especificados, em paralelo.
// Ela devolve os nomes dos arquivos gerados em uma ordem arbitrária,
// ou um erro se algum passo falhar.
func makeThumbnails5(filenames []string) (thumbfiles []string, err error) {
    type item struct {
        thumbfile string
        err        error
    }

    ch := make(chan item, len(filenames))
    for _, f := range filenames {
        go func(f string) {
            var it item
            it.thumbfile, it.err = thumbnail.ImageFile(f)
            ch <- it
        }(f)
    }

    for range filenames {
        it := <-ch
        if it.err != nil {
            return nil, it.err
        }
    }
}
```

```

        thumbfiles = append(thumbfiles, it.thumbfile)
    }
    return thumbfiles, nil
}

```

Nossa versão final de **makeThumbnails** a seguir devolve o número total de bytes ocupados pelos novos arquivos. Diferente das versões anteriores, porém, ela recebe os nomes dos arquivos não como uma fatia, mas por meio de um canal de strings, portanto, não podemos prever o número de iterações do loop.

Para saber quando a última gorrotina terminou (que pode não ser a última a iniciar), devemos incrementar um contador antes que cada gorrotina inicie e decrementá-lo à medida que cada gorrotina termine. Isso exige um tipo especial de contador, um que possa ser manipulado de forma segura a partir de várias gorrotinas e ofereça uma maneira de esperar até que ele seja igual a zero. Esse tipo de contador é conhecido como **sync.WaitGroup**, e o código a seguir mostra como usá-lo:

```

// makeThumbnails6 cria miniaturas para cada arquivo recebido pelo canal.
// Ele devolve o número de bytes ocupados pelos arquivos criados.
func makeThumbnails6(filenames <-chan string) int64 {
    sizes := make(chan int64)
    var wg sync.WaitGroup // número de gorrotinas de trabalho
    for f := range filenames {
        wg.Add(1)
        // gorrotina de trabalho
        go func(f string) {
            defer wg.Done()
            thumb, err := thumbnail.ImageFile(f)
            if err != nil {
                log.Println(err)
                return
            }
            info, _ := os.Stat(thumb) // tudo bem ignorar erro neste caso
            sizes <- info.Size()
        }(f)
    }
    // gorrotina de fechamento
    go func() {
        wg.Wait()
    }()
}

```

```

        close(sizes)
    }()
    var total int64
    for size := range sizes {
        total += size
    }
    return total
}

```

Observe a assimetria nos métodos **Add** e **Done**. **Add**, que incrementa o contador, deve ser chamado antes que as gorrotinas de trabalho iniciem, e não dentro delas; caso contrário, não teríamos certeza de que **Add** *acontece antes* de a gorrotina de “fechamento” (closer) chamar **Wait**. Além disso, **Add** aceita um parâmetro, mas **Done** não; é equivalente a **Add(-1)**. Usamos **defer** para garantir que o contador seja decrementado mesmo no caso de erro. A estrutura do código anterior é um padrão comum e idiomático para looping em paralelo quando não conhecemos o número de iterações.

O canal **sizes** transporta cada tamanho de arquivo de volta para a gorrotina principal, que os recebe usando um loop **range**, e calcula a soma. Observe como criamos uma gorrotina de fechamento que espera as gorrotinas de trabalho terminarem antes de fechar o canal **sizes**. Essas duas operações, esperar e fechar, devem ser concorrentes com o loop em **sizes**. Considere as alternativas: se a operação de espera fosse colocada na gorrotina principal antes do loop, ela jamais terminaria, e se fosse colocada depois dele, seria inacessível, pois, sem que haja nada para fechar o canal, o loop jamais terminaria.

A figura 8.5 mostra a sequência de eventos na função **makeThumbnaïls6**. As linhas verticais representam as gorrotinas. Os segmentos finos indicam que a gorrotina dorme, enquanto os segmentos espessos indicam atividade. As setas diagonais indicam eventos que sincronizam uma gorrotina com outra. O tempo flui para baixo. Observe como a gorrotina principal gasta a maior parte de seu tempo dormindo no loop **range**, esperando que uma gorrotina de trabalho envie um valor ou que a gorrotina de fechamento feche o canal.

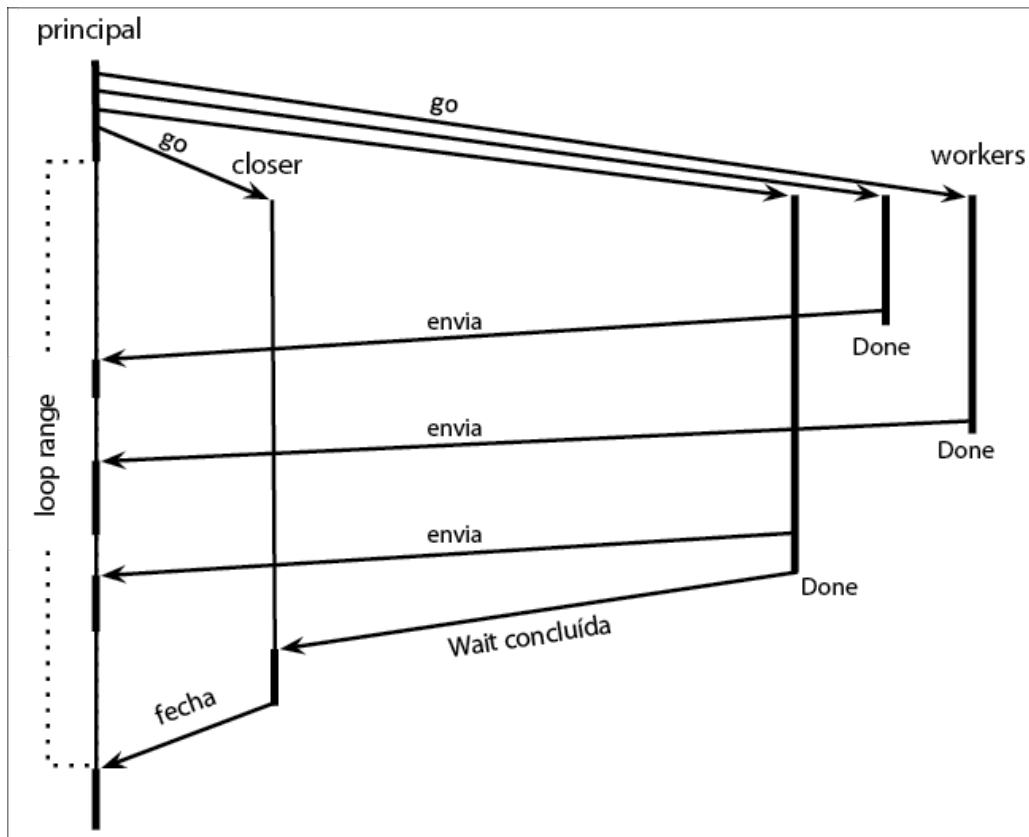


Figura 8.5 – A sequência de eventos em *makeThumbnails6*.

Exercício 8.4: Modifique o servidor **reverb2** para que use um **sync.WaitGroup** por conexão para contar o número de gorrotinas **echo** ativas. Quando atingir zero, feche a metade de escrita da conexão TCP conforme descrito no exercício 8.3. Certifique-se de que seu cliente **netcat3** modificado daquele exercício espere os ecos finais dos vários “gritos” concorrentes, mesmo depois que a entrada-padrão tiver sido fechada.

Exercício 8.5: Tome um programa sequencial existente limitado por CPU (CPU-bound), por exemplo, o programa de Mandelbrot da seção 3.3 ou de processamento de superfícies 3D da seção 3.2, e execute seu loop principal em paralelo usando canais para comunicação. Quão mais rápido ele executa em um computador com vários processadores? Qual é o número ideal de gorrotinas a ser usado?

8.6 Exemplo: Web crawler concorrente

Na seção 5.6, criamos um web crawler simples que explorava o grafo de links da web em ordem de largura (breadth-first order). Nesta seção, vamos deixá-lo concorrente para que chamadas independentes a `crawl` possam explorar o paralelismo de E/S disponível na web. A função `crawl` permanece exatamente igual àquela em gopl.io/ch5/findlinks3:

gopl.io/ch8/crawl1

```
func crawl(url string) []string {
    fmt.Println(url)
    list, err := links.Extract(url)
    if err != nil {
        log.Print(err)
    }
    return list
}
```

A função principal lembra `breadthFirst` (seção 5.6). Como antes, uma lista de trabalho (worklist) armazena a fila de itens que deve ser processada, em que cada item é uma lista de URLs para percorrer, porém, desta vez, no lugar de representar a fila usando uma fatia, usaremos um canal. Cada chamada a `crawl` ocorre em sua própria gorrotina, e os links descobertos são enviados de volta à lista de trabalho.

```
func main() {
    worklist := make(chan []string)

    // Começa com os argumentos da linha de comando.
    go func() { worklist <- os.Args[1:] }()

    // Faz crawling da web de forma concorrente.
    seen := make(map[string]bool)
    for list := range worklist {
        for _, link := range list {
            if !seen[link] {
                seen[link] = true
                go func(link string) {
                    worklist <- crawl(link)
                }(link)
            }
        }
    }
}
```

```
}  
}
```

Observe que a gorrotina que faz crawling recebe `link` como um parâmetro explícito para evitar o problema de captura de variável de loop que vimos inicialmente na seção 5.6.1. Observe também que o envio inicial dos argumentos da linha de comando à lista de trabalho deve executar em sua própria gorrotina para evitar *deadlock* – uma situação de bloqueio em que tanto a gorrotina principal quanto a gorrotina crawler tentam fazer envios uma para a outra, enquanto nenhuma delas recebe. Uma solução alternativa seria usar um canal com buffer.

O crawler agora é altamente concorrente e exibe uma enxurrada de URLs, mas ele tem dois problemas. O primeiro se manifesta como mensagens de erro no log após alguns segundos de funcionamento:

```
$ go build gopl.io/ch8/crawl1  
$ ./crawl1 http://gopl.io/  
http://gopl.io/  
https://golang.org/help/  
https://golang.org/doc/  
https://golang.org/blog/  
...  
2015/07/15 18:22:12 Get ...: dial tcp: lookup blog.golang.org: no such host  
2015/07/15 18:22:12 Get ...: dial tcp 23.21.222.120:443: socket:  
too many open files  
...
```

A mensagem de erro inicial é uma informação surpreendente de uma falha de busca de DNS para um domínio confiável. A mensagem de erro subsequente revela a causa: o programa criou tantas conexões de rede ao mesmo tempo que excedeu o limite para o número de arquivos abertos por processo, fazendo operações como buscas de DNS e chamadas a `net.Dial` começarem a falhar.

O programa é paralelo *demais*. Paralelismo sem limites raramente é uma boa ideia, pois há sempre um fator limitante no sistema, como o número de CPUs para tarefas limitadas por processamento (compute-bound), o número de eixos e cabeçotes para operações de E/S em discos locais, a largura de banda de rede para downloads de streaming ou a capacidade

de servir de um web service. A solução é limitar o número de usos paralelos de recursos para que corresponda ao nível de paralelismo disponível. Uma maneira simples de fazer isso em nosso exemplo é garantir que não mais do que n chamadas a `links.Extract` estejam ativas ao mesmo tempo, em que n seja confortavelmente menor que o limite de descritores de arquivos – por exemplo, vinte. Isso é análogo à maneira como um segurança em uma boate noturna lotada aceita clientes somente à medida que outros clientes saiam.

Podemos limitar o paralelismo usando um canal com buffer de capacidade n para modelar uma primitiva de concorrência chamada *semáforo contador* (counting semaphore). Conceitualmente, cada uma das n posições vazias no buffer do canal representa um token que permite que seu dono prossiga. Enviar um valor ao canal faz um token ser adquirido e receber um valor do canal libera um token, liberando uma nova posição. Isso garante que no máximo n envios possam ocorrer sem que haja uma recepção no meio. (Embora possa ser mais intuitivo tratar posições *preenchidas* no buffer do canal como tokens, usar posições livres evita a necessidade de preencher o buffer do canal após criá-lo.) Como o tipo do elemento do canal não é importante, usaremos `struct{}`, cujo tamanho é zero.

Vamos reescrever a função `crawl` de modo que a chamada a `links.Extract` esteja cercada por operações de aquisição e liberação de um token, garantindo assim que, no máximo, vinte chamadas a ela estejam ativas ao mesmo tempo. É uma boa prática manter as operações de semáforo tão próximas quanto possível da operação de E/S que elas administram.

gopl.io/ch8/crawl2

```
// tokens é um semáforo contador usado para
// impor um limite de 20 requisições concorrentes.
var tokens = make(chan struct{}, 20)

func crawl(url string) []string {
    fmt.Println(url)
    tokens <- struct{}{} // adquire um token
```



```

    list, err := links.Extract(url)
    <-tokens // libera o token
    if err != nil {
        log.Print(err)
    }
    return list
}

```

O segundo problema é que o programa nunca termina, mesmo depois de descobrir todos os links acessíveis a partir dos URLs iniciais. (É claro que é improvável que você vá perceber esse problema, a menos que escolha os URLs iniciais cuidadosamente ou implemente o recurso de limitação de profundidade do exercício 8.6.) Para que o programa termine, precisamos sair do loop principal quando a lista de trabalho estiver vazia *e* não houver mais gorrotinas de crawling ativas.

```

func main() {
    worklist := make(chan []string)
    var n int // número de envios pendentes para a lista de trabalho

    // Começa com os argumentos da linha de comando.
    n++
    go func() { worklist <- os.Args[1:] }()

    // Faz crawling da web de forma concorrente.
    seen := make(map[string]bool)
    for ; n > 0; n-- {
        list := <-worklist
        for _, link := range list {
            if !seen[link] {
                seen[link] = true
                n++
                go func(link string) {
                    worklist <- crawl(link)
                }(link)
            }
        }
    }
}

```

Nessa versão, o contador `n` monitora o número de envios para a lista de trabalho que ainda estão por acontecer. Sempre que soubermos que um item precisa ser enviado à lista de trabalho, incrementamos `n`, uma vez

antes de enviar os argumentos da linha de comando iniciais e novamente sempre que iniciamos uma gorrotina de crawler. O loop principal termina quando `n` atinge zero, pois não há mais trabalho a ser feito.

Agora o crawler concorrente executa aproximadamente vinte vezes mais rápido que o crawler em largura da seção 5.6, sem erros, e termina corretamente se sua tarefa terminar.

O problema a seguir mostra uma solução alternativa para o problema de concorrência excessiva. Esta versão usa a função `crawl` original sem semáforo contador, mas a chama a partir de uma das vinte gorrotinas de crawler de longa duração, garantindo assim que, no máximo, vinte requisições HTTP estejam ativas de forma concorrente.

gopl.io/ch8/crawl3

```
func main() {
    worklist := make(chan []string) // listas de URLs, pode haver duplicações
    unseenLinks := make(chan string) // URLs sem duplicação

    // Adiciona os argumentos da linha de comando à lista de trabalho.
    go func() { worklist <- os.Args[1:] }()

    // Cria 20 gorrotinas crawler para buscar cada link não visto.
    for i := 0; i < 20; i++ {
        go func() {
            for link := range unseenLinks {
                foundLinks := crawl(link)
                go func() { worklist <- foundLinks }()
            }
        }()
    }

    // A gorrotina principal remove os itens duplicados da lista de trabalho
    // e envia os itens não vistos para os crawlers.
    seen := make(map[string]bool)
    for list := range worklist {
        for _, link := range list {
            if !seen[link] {
                seen[link] = true
                unseenLinks <- link
            }
        }
    }
}
```

}

As gorrotinas de crawler são todas alimentadas pelo mesmo canal, **unseenLinks**. A gorrotina principal é responsável pela remoção de itens duplicados recebidos da lista de trabalho e pelo envio de cada item não visto pelo canal **unseenLinks** para uma gorrotina de crawler.

O mapa **seen** está *confinado* na gorrotina principal, isto é, ele pode ser acessado somente por essa gorrotina. Como em outras formas de ocultação de informações, o confinamento nos ajuda a avaliar se um programa está correto. Por exemplo, variáveis locais não podem ser mencionadas pelo nome a partir de fora da função em que são declaradas; variáveis que não escapam (seção 2.3.4) de uma função não podem ser acessadas de fora dessa função; campos encapsulados de um objeto não podem ser acessados, exceto pelos métodos desse objeto. Em todos os casos, ocultar informações ajuda a limitar interações indesejadas entre partes do programa.

Links encontrados por **crawl** são enviados à lista de trabalho a partir de uma gorrotina dedicada para evitar deadlock.

Para economizar espaço, não tratamos o problema do término nesse exemplo.

Exercício 8.6: Acrescente uma limitação de profundidade no crawler concorrente, ou seja, se o usuário definir **-depth=3**, apenas os URLs acessíveis por no máximo três links serão buscados.

Exercício 8.7: Escreva um programa concorrente que crie um espelho local de um site da web, buscando cada página acessível e gravando-a em um diretório no disco local. Somente páginas do domínio original (por exemplo, *golang.org*) devem ser buscadas. URLs em páginas espelhadas devem ser alterados conforme for necessário para que se refiram à página espelhada, e não ao original.

8.7 Multiplexando com select

O programa a seguir faz a contagem regressiva para o lançamento de um

foguete. A função `time.Tick` devolve um canal pelo qual ela envia eventos periodicamente, agindo como um metrônomo. O valor de cada evento é um timestamp, mas raramente ele será tão interessante quando o próprio fato de seu envio.

gopl.io/ch8/countdown1

```
func main() {
    fmt.Println("Commencing countdown.")
    tick := time.Tick(1 * time.Second)
    for countdown := 10; countdown > 0; countdown-- {
        fmt.Println(countdown)
        <-tick
    }
    launch()
}
```

Vamos agora acrescentar a capacidade de abortar a sequência de lançamento pressionando a tecla return durante a contagem regressiva. Em primeiro lugar, iniciamos uma gorrotina que tenta ler um único byte da entrada-padrão e, se for bem-sucedida, envia um valor em um canal chamado `abort`.

gopl.io/ch8/countdown2

```
abort := make(chan struct{})
go func() {
    os.Stdin.Read(make([]byte, 1)) // lê um único byte
    abort <- struct{}{}
}()
```

Agora cada iteração do loop de contagem regressiva precisa esperar que um evento chegue em um dos dois canais: o canal de contagem se tudo correr bem (“nominal” no jargão da NASA) ou um evento para abortar se houver uma “anomalia”. Não podemos simplesmente receber de cada canal, pois a operação executada antes, qualquer que seja, ficará bloqueada até ser concluída. Precisamos *multiplexar* essas operações e para isso, precisamos de uma *instrução de seleção*.

```
select {
case <-ch1:
    // ...
```

```

case x := <-ch2:
    // ...usa x...
case ch3 <- y:
    // ...
default:
    // ...
}

```

O formato geral de uma instrução de seleção foi mostrado acima. Assim como uma instrução `switch`, ela tem alguns casos e um **default** opcional. Cada caso especifica uma *comunicação* (uma operação de envio ou de recepção em algum canal) e um bloco de instruções associado. Uma expressão de recepção pode aparecer sozinha, como no primeiro caso, ou em uma declaração curta de variável, como no segundo. Essa última forma permite referenciar o valor recebido.

Um **select** espera até uma comunicação para algum caso estar pronta para prosseguir. Ele então realiza essa comunicação e executa as instruções associadas ao caso; as outras comunicações não acontecem. Um **select** sem casos, **select{}**, espera indefinidamente.

Vamos retornar ao nosso programa de lançamento de foguete. A função **time.After** devolve imediatamente um canal e inicia uma nova gorrotina que envia um único valor por esse canal após o tempo especificado. A instrução de seleção a seguir espera até o primeiro de dois eventos chegar, seja um evento para abortar ou o evento que indica que dez segundos se passaram. Se dez segundos se passaram sem evento de abortar, o lançamento prossegue.

```

func main() {
    // ...cria o canal abort...

    fmt.Println("Commencing countdown. Press return to abort.")
    select {
    case <-time.After(10 * time.Second):
        // Não faz nada.
    case <-abort:
        fmt.Println("Launch aborted!")
        return
    }
    launch()
}

```

```
}
```

O exemplo seguinte é mais sutil. O canal `ch`, cujo tamanho do buffer é um, fica alternadamente cheio e então vazio; portanto, apenas um dos casos pode prosseguir, seja o envio quando `i` é par, seja a recepção quando `i` é ímpar. Ele sempre exibe `0 2 4 6 8`.

```
ch := make(chan int, 1)
for i := 0; i < 10; i++ {
    select {
    case x := <-ch:
        fmt.Println(x) // "0" "2" "4" "6" "8"
    case ch <- i:
    }
}
```

Se vários casos estiverem prontos, **select** escolhe um aleatoriamente, o que garante que todo canal tenha a mesma chance de ser selecionado. Aumentar o tamanho do buffer no exemplo anterior faz sua saída ser não determinística, pois, quando o buffer não está nem cheio nem vazio, a instrução de seleção decide no “cara ou coroa”, falando de modo figurativo.

Vamos fazer nosso programa de lançamento exibir a contagem regressiva. A instrução de seleção a seguir faz cada iteração do loop esperar até um segundo por um evento de abortar, mas não mais que isso.

gopl.io/ch8/countdown3

```
func main() {
    // ...cria o canal abort...

    fmt.Println("Commencing countdown. Press return to abort.")
    tick := time.Tick(1 * time.Second)
    for countdown := 10; countdown > 0; countdown-- {
        fmt.Println(countdown)
        select {
        case <-tick:
            // Não faz nada.
        case <-abort:
            fmt.Println("Launch aborted!")
            return
        }
    }
}
```

```
    launch()
}
```

A função `time.Tick` comporta-se como se criasse uma gorrotina que chama `time.Sleep` em um loop, enviando um evento sempre que acorda. Quando a função de contagem regressiva anterior retorna, ela para de receber eventos de `tick`, mas a gorrotina de contagem continua presente, tentando em vão fazer um envio por um canal do qual nenhuma gorrotina está recebendo – um *vazamento de gorrotina* (goroutine leak, na seção 8.4.4).

A função `Tick` é conveniente, mas é apropriada somente quando os tiques de contagem são necessários durante todo o tempo de vida da aplicação. Caso contrário, devemos usar este padrão:

```
ticker := time.NewTicker(1 * time.Second)
<-ticker.C // recebe do canal de tiques
ticker.Stop() // faz a gorrotina que gera tiques terminar
```

Às vezes, queremos tentar enviar ou receber por um canal, mas evitar um bloqueio se o canal não estiver pronto – uma comunicação *não bloqueante* (non-blocking). Uma instrução de seleção pode fazer isso também. Um `select` pode ter um `default` que especifica o que deve ser feito quando nenhuma das demais comunicações pode prosseguir imediatamente.

A instrução de seleção a seguir recebe um valor do canal `abort` se houver um para receber; caso contrário, não faz nada. Essa é uma operação de recepção não bloqueante; fazer isso repetidamente chama-se fazer *polling* de um canal.

```
select {
case <-abort:
    fmt.Printf("Launch aborted!\n")
    return
default:
    // não faz nada.
}
```

O valor zero de um canal é `nil`. Talvez seja surpreendente que canais `nil` às vezes sejam úteis. Como as operações de envio e de recepção em um canal `nil` são bloqueadas para sempre, um caso em uma instrução de

seleção cujo canal seja nil jamais será selecionado. Isso nos permite usar `nil` para habilitar ou desabilitar casos que correspondam a recursos como tratamento de timeouts ou cancelamento, respondendo a outros eventos de entrada ou gerando uma saída. Veremos um exemplo na próxima seção.

Exercício 8.8: Usando uma instrução de seleção, adicione um timeout ao servidor de eco da seção 8.3 de modo que ele desconecte qualquer cliente que não grite nada durante dez segundos.

8.8 Exemplo: Travessia concorrente de diretório

Nesta seção, criaremos um programa que informa o uso de disco de um ou mais diretórios especificados na linha de comando, como o comando `du` do Unix. A maior parte de seu trabalho é feita pela função `walkDir` a seguir, que enumera as entradas do diretório `dir` usando a função auxiliar `dirents`.

gopl.io/ch8/du1

```
// walkDir percorre recursivamente a árvore de arquivos cuja raiz é dir
// e envia o tamanho de cada arquivo encontrado para fileSizes.
func walkDir(dir string, fileSizes chan<- int64) {
    for _, entry := range dirents(dir) {
        if entry.IsDir() {
            subdir := filepath.Join(dir, entry.Name())
            walkDir(subdir, fileSizes)
        } else {
            fileSizes <- entry.Size()
        }
    }
}

// dirents devolve as entradas do diretório dir.
func dirents(dir string) []os.FileInfo {
    entries, err := ioutil.ReadDir(dir)
    if err != nil {
        fmt.Fprintf(os.Stderr, "du1: %v\n", err)
        return nil
    }
    return entries
}
```



```
}
```

A função `ioutil.ReadDir` devolve uma fatia de `os.FileInfo` – a mesma informação que uma chamada a `os.Stat` devolve para um único arquivo. Para cada subdiretório, `walkDir` chama a si mesma recursivamente e, para cada arquivo, `walkDir` envia uma mensagem pelo canal `fileSizes`. A mensagem é o tamanho do arquivo em bytes.

A função principal, mostrada a seguir, usa duas gorrotinas. A gorrotina em background chama `walkDir` para cada diretório especificado na linha de comando e, por fim, fecha o canal `fileSizes`. A gorrotina principal calcula a soma dos tamanhos dos arquivos que recebe pelo canal e, por fim, exibe o total.

```
// O comando du1 calcula o uso de disco dos arquivos em um diretório.
package main

import (
    "flag"
    "fmt"
    "io/ioutil"
    "os"
    "path/filepath"
)

func main() {
    // Determina os diretórios iniciais.
    flag.Parse()
    roots := flag.Args()
    if len(roots) == 0 {
        roots = []string{"."}
    }

    // Percorre a árvore de arquivos.
    fileSizes := make(chan int64)
    go func() {
        for _, root := range roots {
            walkDir(root, fileSizes)
        }
        close(fileSizes)
    }()
    // Exibe o resultado.
    var nfiles, nbytes int64
    for size := range fileSizes {
```

```

        nfiles++
        nbytes += size
    }
    printDiskUsage(nfiles, nbytes)
}

func printDiskUsage(nfiles, nbytes int64) {
    fmt.Printf("%d files %.1f GB\n", nfiles, float64(nbytes)/1e9)
}

```

Esse programa faz uma pausa demorada antes de exibir seu resultado:

```

$ go build gopl.io/ch8/du1
$ ./du1 $HOME /usr /bin /etc
213201 files   62.7 GB

```

O programa seria melhor se nos mantivesse informado de seu progresso. Contudo, simplesmente mover a chamada a **printDiskUsage** para o loop faria com que milhares de linhas de saída fossem exibidas.

A variante de **du** a seguir exibe o total periodicamente, mas somente se a flag **-v** for especificada, pois nem todos os usuários vão querer ver as mensagens de progresso. A gorrotina em background que percorre **roots** em um loop permanece inalterada. A gorrotina principal agora usa um gerador de tiques para gerar eventos a cada 500 ms e uma instrução de seleção para esperar por uma mensagem com o tamanho de um arquivo, caso em que o total é atualizado, ou um evento de tique, caso em que o total atual é exibido. Se a flag **-v** não for especificada, o canal **tick** permanece como nil e seu caso em **select** é efetivamente desabilitado.

gopl.io/ch8/du2

```

var verbose = flag.Bool("v", false, "show verbose progress messages")

func main() {
    // ...inicia a gorrotina em background...

    // Exibe o resultado periodicamente.
    var tick <-chan time.Time
    if *verbose {
        tick = time.Tick(500 * time.Millisecond)
    }
    var nfiles, nbytes int64
loop:
    for {

```

```

select {
case size, ok := <-fileSizes:
    if !ok {
        break loop // fileSizes foi fechado
    }
    nfiles++
    nbytes += size
case <-tick:
    printDiskUsage(nfiles, nbytes)
}
}
printDiskUsage(nfiles, nbytes) // total final
}

```

Como o programa não usa mais um loop **range**, o primeiro caso de **select** deve testar explicitamente se o canal **fileSizes** foi fechado, usando a forma de dois resultados da operação de recepção. Se o canal foi fechado, o programa sai do loop. A instrução **break** com rótulo sai tanto de **select** quanto do loop **for**; um **break** sem rótulo sairia somente de **select**, fazendo o loop iniciar a próxima iteração.

Agora o programa fornece atualizações periódicas:

```

$ go build gopl.io/ch8/du2
$ ./du2 -v $HOME /usr /bin /etc
28608 files 8.3 GB
54147 files 10.3 GB
93591 files 15.1 GB
127169 files 52.9 GB
175931 files 62.2 GB
213201 files 62.7 GB

```

No entanto ele ainda demora muito para terminar. Não há motivos para todas as chamadas a **walkDir** não poderem ser feitas concorrentemente, explorando assim o paralelismo do sistema de disco. A terceira versão de **du** a seguir cria uma nova gorrotina para cada chamada a **walkDir**. Ela usa um **sync.WaitGroup** (seção 8.5) para contar o número de chamadas a **walkDir** que continuam ativas, e uma gorrotina de fechamento (closer) para fechar o canal **fileSizes** quando o contador atingir zero.

gopl.io/ch8/du3

```

func main() {

```

```

// ...determina roots...
// Percorre cada raiz da árvore de arquivos em paralelo.
fileSizes := make(chan int64)
var n sync.WaitGroup
for _, root := range roots {
    n.Add(1)
    go walkDir(root, &n, fileSizes)
}
go func() {
    n.Wait()
    close(fileSizes)
}()
// ...loop de seleção...
}

func walkDir(dir string, n *sync.WaitGroup, fileSizes chan<- int64) {
    defer n.Done()
    for _, entry := range dirents(dir) {
        if entry.IsDir() {
            n.Add(1)
            subdir := filepath.Join(dir, entry.Name())
            go walkDir(subdir, n, fileSizes)
        } else {
            fileSizes <- entry.Size()
        }
    }
}

```

Como esse programa cria muitos milhares de gorrotinas em seu ápice, devemos mudar **dirents** para que use um semáforo contador a fim de evitar que arquivos demais sejam abertos ao mesmo tempo, como fizemos para o web crawler na seção 8.6.

```

// sema é um semáforo contador para limitar a concorrência em dirents.
var sema = make(chan struct{}, 20)

// dirents devolve as entradas do diretório dir.
func dirents(dir string) []os.FileInfo {
    sema <- struct{}{} // adquire um token
    defer func() { <-sema }() // libera um token
    // ...

```

Essa versão executa muitas vezes mais rápido que a versão anterior, embora varie muito de um sistema para outro.

Exercício 8.9: Escreva uma versão de **du** que calcule e exiba periodicamente os totais separados para cada um dos diretórios de **root**.

8.9 Cancelamento

Às vezes precisamos instruir uma gorrotina a parar o que está fazendo, por exemplo, em um servidor web fazendo um processamento para um cliente que se desconectou.

Não há nenhuma maneira de uma gorrotina terminar outra diretamente, pois isso deixaria todas as suas variáveis compartilhadas em estados indefinidos. No programa de lançamento de foguete (seção 8.7), enviamos um único valor em um canal chamado **abort**, que a gorrotina de contagem regressiva interpretava como uma solicitação para parar. Mas e se precisássemos cancelar duas gorrotinas, ou um número qualquer delas?

Uma possibilidade poderia ser enviar tantos eventos no canal **abort** quantas forem as gorrotinas a serem canceladas. Se algumas das gorrotinas já tiverem terminado por conta própria, porém, nosso contador seria grande demais e nossos envios ficariam travados. Por outro lado, se essas gorrotinas tiverem gerado outras gorrotinas, nosso contador será pequeno demais e algumas gorrotinas não tomarão conhecimento do cancelamento. Em geral, é difícil saber quantas gorrotinas estão trabalhando para nós em determinado instante. Além do mais, quando uma gorrotina recebe um valor do canal **abort**, ela consome esse valor de modo que outras gorrotinas não o verão. Para fazer o cancelamento, precisamos de um sistema confiável de *broadcast* (difusão) de um evento por um canal de modo que muitas gorrotinas possam vê-lo *quando* ocorrer e possam ver mais tarde que ele *ocorreu*.

Lembre-se de que após um canal ter sido fechado e drenado de todos os valores enviados, operações subsequentes de recepção podem prosseguir imediatamente, produzindo valores zero. Podemos explorar isso para criar um sistema de broadcast: não envie um valor no canal, mas *feche-o*.

Podemos acrescentar o cancelamento no programa **du** da seção anterior com algumas mudanças simples. Inicialmente, criamos um canal de

cancelamento em que nenhum valor é enviado, mas cujo fechamento indica que é hora de o programa parar o que está fazendo. Também definimos uma função utilitária **cancelled** que verifica ou faz *polling* do estado de cancelamento no momento em que é chamada.

gopl.io/ch8/du4

```
var done = make(chan struct{})

func cancelled() bool {
    select {
    case <-done:
        return true
    default:
        return false
    }
}
```

Em seguida, criamos uma gorrotina que lerá da entrada-padrão, que geralmente está conectada ao terminal. Assim que uma entrada é lida (por exemplo, o usuário aperta a tecla return), essa gorrotina faz um broadcast do cancelamento fechando o canal **done**.

```
// Cancela a travessia quando uma entrada é detectada.
go func() {
    os.Stdin.Read(make([]byte, 1)) // lê um único byte
    close(done)
}()
```

Agora precisamos fazer nossas gorrotinas responderem ao cancelamento. Na gorrotina principal, adicionamos um terceiro caso à instrução de seleção que tenta receber do canal **done**. A função retorna se esse caso for selecionado, mas, antes de retornar, ela deve drenar o canal **fileSizes**, descartando todos os valores até o canal ser fechado. Ela faz isso para garantir que qualquer chamada ativa a **walkDir** possa executar até o fim sem ficar travada ao enviar para **fileSizes**.

```
for {
    select {
    case <-done:
        // Drena fileSizes para permitir que gorrotinas existentes terminem.
        for range fileSizes {
            // Não faz nada.
```

```

    }
    return
case size, ok := <-fileSizes:
    // ...
}
}

```

A gorrotina `walkDir` faz polling do status de cancelamento quando inicia e retorna sem fazer nada se o status estiver definido. Isso deixa todas as gorrotinas criadas após o cancelamento não operantes:

```

func walkDir(dir string, n *sync.WaitGroup, fileSizes chan<- int64) {
    defer n.Done()
    if cancelled() {
        return
    }
    for _, entry := range dirents(dir) {
        // ...
    }
}

```

Pode valer a pena fazer polling do status de cancelamento novamente no loop de `walkDir` para evitar a criação de gorrotinas após o evento de cancelamento. O cancelamento envolve uma troca: uma resposta mais rápida geralmente exige mudanças mais intrusivas na lógica de programação. Garantir que nenhuma operação custosa vá ocorrer após o evento de cancelamento pode exigir atualizações em muitos lugares no código, mas, com frequência, a maior parte das vantagens pode ser obtida verificando o cancelamento em alguns pontos importantes.

Um pequeno profiling desse programa mostrou que o gargalo foi a aquisição de um token de semáforo em `dirents`. O `select` a seguir deixa essa operação cancelável e reduz a latência típica de cancelamento do programa, de centenas para dezenas de milissegundos:

```

func dirents(dir string) []os.FileInfo {
    select {
    case sema <- struct{}{}: // adquire um token
    case <-done:
        return nil // cancelado
    }
    defer func() { <-sema }() // libera um token
}

```

```
// ...lê o diretório...  
}
```

Agora, quando o cancelamento ocorre, todas as gorrotinas em background param rapidamente, e a função **main** retorna. É claro que quando **main** retorna um programa termina, portanto, pode ser difícil diferenciar uma função **main** que faça uma limpeza de outra que não faz. Há um truque prático que podemos usar durante os testes: em vez de retornar de **main** no caso de cancelamento, executamos uma chamada a **panic**; então, o runtime fará o dump da pilha de todas as gorrotinas do programa. Se a gorrotina principal for a única restante, é sinal de que ela fez a limpeza. Porém, se houver outras gorrotinas restantes, elas podem não ter sido apropriadamente canceladas ou, quem sabe, foram canceladas, mas o cancelamento é demorado, e talvez valha a pena fazer uma pequena investigação. O dump do pânico geralmente contém informações suficientes para fazer a distinção entre esses casos.

Exercício 8.10: Requisições HTTP podem ser canceladas fechando o canal **Cancel** opcional na estrutura **http.Request**. Modifique o web crawler da seção 8.6 para que aceite cancelamento.

Dica: a função conveniente **http.Get** não dá a oportunidade de personalizar um **Request**. Em vez disso, crie a requisição usando **http.NewRequest**, defina seu campo **Cancel** e, então, faça a requisição chamando **http.DefaultClient.Do(req)**.

Exercício 8.11: Seguindo a abordagem de **mirroredQuery** da seção 8.4.4, implemente uma variante de **fetch** que requisiite vários URL de forma concorrente. Assim que a primeira resposta chegar, cancele as demais requisições.

8.10 Exemplo: Servidor de bate-papo

Encerraremos este capítulo com um servidor de bate-papo (chat) que permite que vários usuários façam broadcast de mensagens de texto uns aos outros. Há quatro tipos de gorrotina nesse programa. Há uma instância para a gorrotina **main** e uma para **broadcaster** e, para cada

conexão com um cliente, há uma gorrotina **handleConn** e uma gorrotina **clientWriter**. O broadcaster é um bom exemplo de como **select** é usado, pois ele deve responder a três tipos diferentes de mensagens.

O papel da gorrotina principal, mostrada a seguir, é ouvir e aceitar conexões de rede dos clientes. Para cada conexão, ela cria uma nova gorrotina **handleConn**, assim como no servidor de eco concorrente que vimos no início deste capítulo.

gopl.io/ch8/chat

```
func main() {
    listener, err := net.Listen("tcp", "localhost:8000")
    if err != nil {
        log.Fatal(err)
    }
    go broadcaster()
    for {
        conn, err := listener.Accept()
        if err != nil {
            log.Print(err)
            continue
        }
        go handleConn(conn)
    }
}
```

A seguir, temos o broadcaster. Sua variável local **clients** registra o conjunto atual dos clientes conectados. A única informação registrada sobre cada cliente é a identidade de seu canal de mensagem de saída, que veremos mais adiante.

```
type client chan<- string // um canal de mensagem de saída
var (
    entering = make(chan client)
    leaving  = make(chan client)
    messages = make(chan string) // todas as mensagens que chegam dos clientes
)

func broadcaster() {
    clients := make(map[client]bool) // todos os clientes conectados
    for {
        select {
```

```

    case msg := <-messages:
        // Broadcast de mensagem de entrada para todos
        // os canais de mensagem de saída dos clientes.
        for cli := range clients {
            cli <- msg
        }
    case cli := <-entering:
        clients[cli] = true
    case cli := <-leaving:
        delete(clients, cli)
        close(cli)
    }
}
}

```

O broadcaster ouve os canais **entering** e **leaving** globais em busca de anúncios de chegadas e partidas de clientes. Quando recebe um desses eventos, ele atualiza o conjunto **clients** e, se o evento foi uma partida, ele fecha o canal de mensagem de saída do cliente. O broadcaster também ouve eventos no canal global **messages**, ao qual cada cliente envia todas as suas mensagens de entrada. Quando o broadcaster recebe um desses eventos, ele faz o broadcast da mensagem a todos os clientes conectados.

Vamos agora dar uma olhada nas gorrotinas para cada cliente. A função **handleConn** cria um novo canal de mensagem de saída para seu cliente e anuncia a chegada desse cliente ao broadcaster por meio do canal **entering**. Então ele lê todas as linhas de texto do cliente, enviando cada linha para o broadcaster por meio do canal global de mensagens que chegam, prefixando cada mensagem com a identidade de quem a enviou. Quando não houver mais nada para ler do cliente, **handleConn** anuncia a partida do cliente por meio do canal **leaving** e fecha a conexão.

```

func handleConn(conn net.Conn) {
    ch := make(chan string) // mensagens de saída do cliente
    go clientWriter(conn, ch)

    who := conn.RemoteAddr().String()
    ch <- "You are " + who
    messages <- who + " has arrived"
    entering <- ch
}

```

```

    input := bufio.NewScanner(conn)
    for input.Scan() {
        messages <- who + ": " + input.Text()
    }
    // NOTA: ignorando erros em potencial de input.Err()

    leaving <- ch
    messages <- who + " has left"
    conn.Close()
}

func clientWriter(conn net.Conn, ch <-chan string) {
    for msg := range ch {
        fmt.Fprintln(conn, msg) // NOTA: ignorando erros de rede
    }
}

```

Além disso, `handleConn` cria uma gorrotina `clientWriter` para cada cliente que recebe mensagens transmitidas para o canal de mensagem de saída e as escreve na conexão de rede do cliente. O loop de escrita do cliente termina quando o broadcaster fecha o canal após receber uma notificação de `leaving`.

A apresentação a seguir mostra o servidor em ação, com dois clientes em janelas separadas no mesmo computador, usando `netcat` para conversar:

```

$ go build gopl.io/ch8/chat
$ go build gopl.io/ch8/netcat3
$ ./chat &
$ ./netcat3
You are 127.0.0.1:64208
127.0.0.1:64211 has arrived
Hi!
127.0.0.1:64208: Hi!
127.0.0.1:64211: Hi yourself.
^C
127.0.0.1:64208 has left

$ ./netcat3
You are 127.0.0.1:64216
127.0.0.1:64211: Welcome.
127.0.0.1:64211 has left

$ ./netcat3
You are 127.0.0.1:64211
127.0.0.1:64216 has arrived
Welcome.
127.0.0.1:64211: Welcome.
^C
127.0.0.1:64216 has left

```

Apesar de hospedar uma sessão de bate-papo para n clientes, esse programa executa $2n+2$ gorrotinas que se comunicam concorrentemente, embora não precise de operações explícitas de travamento (locking) – ver a seção 9.2. O mapa **clients** permanece confinado em uma única gorrotina, portanto, o broadcaster não pode ser acessado de forma concorrente. As únicas variáveis compartilhadas por várias gorrotinas são os canais e as instâncias de **net.Conn**, e ambos são *seguros para concorrência* (concurrency safe). Falaremos mais sobre confinamento, segurança em concorrência e as implicações do compartilhamento de variáveis entre gorrotinas no próximo capítulo.

Exercício 8.12: Faça o broadcaster anunciar o conjunto atual de clientes a cada nova chegada. Isso exige que o conjunto **clients** e os canais **entering** e **leaving** registrem o nome do cliente também.

Exercício 8.13: Faça o servidor de bate-papo desconectar clientes ociosos, por exemplo, aqueles que não enviaram nenhuma mensagem nos últimos cinco minutos. Dica: chamar **conn.Close()** em outra gorrotina desbloqueia chamadas ativas a **Read**, como aquela feita por **input.Scan()**.

Exercício 8.14: Altere o protocolo de rede do servidor de bate-papo de modo que cada cliente forneça seu nome na entrada. Use esse nome no lugar do endereço de rede quando prefixar cada mensagem com a identidade de quem está fazendo o envio.

Exercício 8.15: Falha de qualquer programa cliente em ler dados em tempo hábil, em última instância, faz todos os clientes ficarem travados. Modifique o broadcaster para que ignore uma mensagem em vez de esperar, caso o writer do cliente não esteja pronto para aceitá-la. De modo alternativo, acrescente o uso de buffer no canal de mensagem de saída de cada cliente para que a maioria das mensagens não seja descartada; o broadcaster deve usar um envio não bloqueante nesse canal.

9

Concorrência com variáveis compartilhadas

No capítulo anterior, apresentamos vários programas que usavam gorrotinas e canais para expressar concorrência de forma direta e natural. Contudo, ao fazer isso, tratamos superficialmente alguns problemas importantes e sutis que os programadores devem ter em mente quando escrevem código concorrente.

Neste capítulo, daremos uma olhada mais de perto no funcionamento da concorrência. Em particular, destacaremos alguns dos problemas associados ao compartilhamento de variáveis entre várias gorrotinas, as técnicas de análise para reconhecer esses problemas e os padrões para solucioná-los. Por fim, explicaremos algumas das diferenças técnicas entre gorrotinas e as threads do sistema operacional.

9.1 Condições de concorrência

Em um programa sequencial, isto é, em um programa com apenas uma gorrotina, os passos do programa acontecem na ordem de execução familiar, determinada pela lógica do programa. Por exemplo, em uma sequência de instruções, a primeira acontece antes da segunda, e assim por diante. Em um programa com duas ou mais gorrotinas, os passos em cada gorrotina acontecem na ordem familiar, mas, em geral, não sabemos se um evento x em uma gorrotina acontece antes de um evento y em outra gorrotina ou depois dele, ou simultaneamente. Quando não podemos dizer com certeza que um evento *acontece antes* de outro, então os eventos x e y são *concorrentes*.

Considere uma função que opere corretamente em um programa sequencial. Essa função é *segura para concorrência* (concurrency-safe) se continuar a operar corretamente, mesmo quando chamada de forma

concorrente, isto é, a partir de duas ou mais gorrotinas sem sincronização adicional. Podemos generalizar essa noção para um conjunto de funções que colaboram entre si, por exemplo, os métodos e as operações de um tipo particular. Um tipo é seguro para concorrência se todos os seus métodos e operações acessíveis forem seguros para concorrência.

Podemos deixar um programa seguro para concorrência sem fazer com que todo tipo concreto nesse programa seja seguro para concorrência. Na verdade, tipos seguros para concorrência são a exceção e não a regra, portanto, você deve acessar uma variável de forma concorrente somente se a documentação para seu tipo afirmar que isso é seguro. Evitamos acesso concorrente à maioria das variáveis, seja *confinando-as* em uma única gorrotina, seja preservando uma invariante de *exclusão mútua* (mutual exclusion) em um nível mais alto. Explicaremos esses termos neste capítulo.

Em comparação, espera-se que funções exportadas no nível de pacote *sejam* seguras para concorrência. Como variáveis de nível de pacote não podem ser confinadas em uma única gorrotina, funções que as modificam devem garantir uma exclusão mútua.

Há vários motivos pelos quais uma função pode não operar corretamente quando chamada de forma concorrente, incluindo deadlock, livelock¹ e inanição de recursos (resource starvation)². Não temos espaço para discutir todos eles, portanto, focaremos no mais importante: a *condição de corrida* (race condition).

Uma condição de corrida é uma situação em que o programa não oferece o resultado correto para alguns entrelaçamentos de operações de várias gorrotinas. Condições de concorrência são nocivas porque podem permanecer latentes em um programa e não surgirem com frequência, talvez apenas quando houver muita carga ou quando determinados compiladores, plataformas ou arquiteturas forem usados. Isso as torna difíceis de reproduzir e de diagnosticar.

É tradicional explicar a gravidade das condições de concorrência por meio da metáfora de perda financeira. Assim, vamos considerar um programa

simples de gerenciamento de conta bancária:

```
// O pacote bank implementa um banco com apenas uma conta
package bank

var balance int

func Deposit(amount int) { balance = balance + amount }

func Balance() int { return balance }
```

(Poderíamos ter escrito o corpo da função **Deposit** como **balance += amount**, que é equivalente, mas a forma mais longa simplificará a explicação.)

Para um programa tão trivial quanto esse, podemos ver, à primeira vista, que qualquer sequência de chamadas a **Deposit** e a **Balance** fornecerão a resposta correta, ou seja, **Balance** informará a soma de todos os valores depositados anteriormente. No entanto, se chamarmos essas funções de forma concorrente, e não em uma sequência, não teremos mais garantia de que **Balance** fornecerá a resposta correta. Considere as duas gorrotinas a seguir, que representam duas transações em uma conta bancária conjunta:

```
// Alice:
go func() {
    bank.Deposit(200)           // A1
    fmt.Println("=", bank.Balance()) // A2
}()
// Bob:
go bank.Deposit(100)           // B
```

Alice deposita 200 dólares e então verifica seu saldo, enquanto Bob deposita 100 dólares. Como os passos **A1** e **A2** ocorrem de forma concorrente a **B**, não podemos prever a ordem em que eles acontecem. Intuitivamente, pode parecer que há apenas três ordens possíveis, que chamaremos de “Alice primeiro”, “Bob primeiro” e “Alice/Bob/Alice”. A tabela a seguir mostra o valor da variável **balance** após cada passo. As strings entre aspas representam o saldo exibido.

Alice primeiro	Bob primeiro	Alice/Bob/Alice
0	0	0
A1 200	B 100	A1 200

A2 "= 200"
B 300

A1 300
A2 "= 300"

B 300
A2 "= 300"

Em todos os casos, o saldo final é de 300 dólares. A única variação é se o saldo exibido de Alice inclui a transação de Bob ou não, mas os clientes estarão satisfeitos, de qualquer maneira.

Porém, essa intuição está incorreta. Há um quarto resultado possível, em que o depósito de Bob ocorre no meio do depósito de Alice, depois que o saldo é lido (**balance + amount**), mas antes de ele ser atualizado (**balance = ...**), fazendo a transação de Bob desaparecer. Isso ocorre porque a operação de depósito de Alice, **A1**, é na verdade uma sequência de duas operações: uma leitura e uma escrita – vamos chamá-las de **A1r** e **A1w**. Aqui está o entrelaçamento problemático:

Concorrência nos dados

	0	
A1r	0	... = balance + amount
B	100	
A1w	200	balance = ...
A2	"= 200"	

Depois de **A1r**, a expressão **balance + amount** é avaliada com 200, portanto, esse é o valor escrito durante **A1w**, apesar do depósito no meio. O saldo final é de apenas 200 dólares. O banco está 100 dólares mais rico à custa de Bob.

Esse programa contém um tipo particular de condição de corrida chamada *concorrência de dados* (data race). Uma concorrência de dados ocorre sempre que duas gorrotinas acessam a mesma variável concorrentemente e pelo menos um dos acessos é uma escrita.

A situação fica mais confusa ainda se a concorrência de dados envolver uma variável de um tipo que seja maior que uma única palavra de computador, como uma interface, uma string ou uma fatia. O código a seguir atualiza **x** de forma concorrente com duas fatias de tamanhos diferentes:

```
var x []int
go func() { x = make([]int, 10) }()
go func() { x = make([]int, 1000000) }()
```



```
x[999999] = 1 // NOTA: comportamento indefinido; possível corrupção  
// de memória!
```

O valor de `x` na última instrução não está definido; poderia ser `nil`, uma fatia de tamanho 10 ou uma fatia de tamanho 1.000.000. Porém, lembre-se de que há três partes em uma fatia: o ponteiro, o tamanho e a capacidade. Se o ponteiro vier da primeira chamada a `make` e o tamanho vier da segunda, `x` seria uma aberração – uma fatia cujo tamanho nominal é 1.000.000, mas cujo array subjacente tem apenas 10 elementos. Nessa eventualidade, armazenar um dado no elemento 999.999 encobriria uma posição distante qualquer de memória, com consequências que são impossíveis de prever e difíceis de depurar e localizar. Esse campo minado semântico é conhecido como *comportamento indefinido* (undefined behavior) e é bem-conhecido de programadores C; por sorte, raramente ele é tão problemático em Go quanto em C.

Até mesmo a noção de que um programa concorrente é um entrelaçamento de vários programas sequenciais é uma falsa intuição. Como veremos na seção 9.4, concorrência de dados pode ter resultados ainda mais estranhos. Muitos programadores – mesmo alguns muito inteligentes – ocasionalmente oferecerão justificativas para concorrências de dados conhecidas em seus programas: “o custo da exclusão mútua é alto demais”, “essa lógica é apenas para logging”, “não me importo de perder algumas mensagens”, e assim por diante. A ausência de problemas em um dado compilador e plataforma pode lhes dar uma falsa confiança. Uma boa regra geral é: *não existe algo como uma concorrência de dados benigna*. Então, como evitamos concorrência de dados em nossos programas?

Repetiremos a definição, pois ela é muito importante: uma concorrência de dados ocorre sempre que duas gorrotinas acessam a mesma variável concorrentemente e pelo menos um dos acessos é uma escrita. Da definição, segue-se que há três maneiras de evitar uma concorrência de dados.

A primeira maneira é não escrever na variável. Considere o mapa a seguir, que é preenchido em modo lazy (preguiçoso), à medida que cada chave é

solicitada pela primeira vez. Se **Icon** for chamada sequencialmente, o programa funcionará de forma correta, mas se **Icon** for chamada de forma concorrente, haverá uma concorrência de dados para acessar o mapa.

```
var icons = make(map[string]image.Image)

func loadIcon(name string) image.Image

// NOTA: não é seguro para concorrência!
func Icon(name string) image.Image {
    icon, ok := icons[name]
    if !ok {
        icon = loadIcon(name)
        icons[name] = icon
    }
    return icon
}
```

Se, alternativamente, inicializarmos o mapa com todas as entradas necessárias antes de criar gorrotinas adicionais e jamais o modificarmos de novo, qualquer gorrotina poderá chamar **Icon** de forma segura concorrentemente, pois cada uma delas apenas lerá o mapa.

```
var icons = map[string]image.Image{
    "spades.png":  loadIcon("spades.png"),
    "hearts.png":  loadIcon("hearts.png"),
    "diamonds.png": loadIcon("diamonds.png"),
    "clubs.png":   loadIcon("clubs.png"),
}

// Seguro para concorrência.
func Icon(name string) image.Image { return icons[name] }
```

No exemplo anterior, a atribuição à variável **icons** ocorre durante a inicialização do pacote, que *acontece antes* de a função **main** do programa começar a executar. Depois de inicializada, **icons** jamais será modificada. Estruturas de dados que nunca são modificadas ou são imutáveis são inerentemente seguras para concorrência e não precisam de sincronização. Porém, é óbvio que não podemos usar essa abordagem se as atualizações forem essenciais, como no caso da conta bancária.

A segunda maneira de evitar uma concorrência de dados é evitar acessar a

variável a partir de várias gorrotinas. Essa é a abordagem usada por muitos dos programas do capítulo anterior. Por exemplo, a gorrotina principal no web crawler concorrente (seção 8.6) é a única gorrotina que acessa o mapa **seen**, e a gorrotina **broadcaster** no servidor de bate-papo (seção 8.10) é a única gorrotina que acessa o mapa **clients**. Essas variáveis estão *confinadas* em uma única gorrotina.

Como outras gorrotinas não podem acessar a variável diretamente, elas devem usar um canal para enviar uma solicitação à gorrotina responsável pelo confinamento para consultar ou atualizar a variável. É isso que quer dizer o mantra de Go: “Não se comunique compartilhando memória; em vez disso, compartilhe memória se comunicando”. Uma gorrotina que administra o acesso a uma variável confinada usando requisições de canal chama-se *gorrotina monitora* (monitor goroutine) dessa variável. Por exemplo, a gorrotina **broadcaster** monitora o acesso ao mapa **clients**.

Eis o exemplo do banco reescrito com a variável **balance** confinada a uma gorrotina monitora chamada **teller**:

gopl.io/ch9/bank1

```
// O pacote bank oferece um banco seguro para concorrência, com uma conta.
package bank

var deposits = make(chan int) // envia um valor para depósito
var balances = make(chan int) // recebe o saldo

func Deposit(amount int) { deposits <- amount }
func Balance() int       { return <- balances }

func teller() {
    var balance int // balance está confinada na gorrotina teller
    for {
        select {
        case amount := <-deposits:
            balance += amount
        case balances <- balance:
        }
    }
}

func init() {
    go teller() // inicia a gorrotina monitora
}
```

Mesmo quando uma variável não pode ser confinada em uma única gorrotina durante todo o seu tempo de vida, o confinamento ainda pode ser uma solução para o problema de acesso concorrente. Por exemplo, é comum compartilhar uma variável entre gorrotinas em um pipeline, passando seu endereço de um estágio para o próximo por meio de um canal. Se cada estágio do pipeline evitar acessar a variável após enviá-la para o próximo estágio, então todos os acessos à variável serão sequenciais. Com efeito, a variável estará confinada em um estágio do pipeline e então estará confinada no próximo estágio, e assim sucessivamente. Essa organização às vezes é chamada de *confinamento serial* (serial confinement).

No exemplo a seguir, os **Cakes** estão confinados serialmente, primeiro na gorrotina **baker** e depois na gorrotina **icer**:

```
type Cake struct{ state string }
func baker(cooked chan<- *Cake) {
    for {
        cake := new(Cake)
        cake.state = "cooked"
        cooked <- cake // baker jamais toca neste cake novamente
    }
}
func icer(iced chan<- *Cake, cooked <- chan *Cake) {
    for cake := range cooked {
        cake.state = "iced"
        iced <- cake // icer jamais toca neste cake novamente
    }
}
```

A terceira maneira de evitar uma concorrência de dados é permitir que várias gorrotinas acessem a variável, mas somente uma de cada vez. Essa abordagem é conhecida como *exclusão mútua* (mutual exclusion) e será o assunto da próxima seção.

Exercício 9.1: Acrescente uma função `Withdraw(amount int) bool` ao programa `gopl.io/ch9/bank1`. O resultado deve informar se a transação foi bem-sucedida ou se falhou devido a um saldo insuficiente. A mensagem enviada à gorrotina monitora deve conter tanto o valor a ser

sacado quanto um novo canal por meio do qual a gorrotina monitora possa enviar o resultado booleano de volta a `Withdraw`.

9.2 Exclusão mútua: `sync.Mutex`

Na seção 8.6, utilizamos um canal com buffer como um *semáforo contador* para garantir que não mais de vinte gorrotinas fizessem requisições HTTP simultaneamente. A partir da mesma ideia, podemos usar um canal de capacidade um para garantir que, no máximo, uma gorrotina acesse uma variável compartilhada ao mesmo tempo. Um semáforo que conte apenas até um chama-se *semáforo binário* (binary semaphore).

gopl.io/ch9/bank2

```
var (
    sema = make(chan struct{}, 1) // um semáforo binário que protege balance
    balance int
)

func Deposit(amount int) {
    sema <- struct{}{} // adquire o token
    balance = balance + amount
    <-sema // libera o token
}

func Balance() int {
    sema <- struct{}{} // adquire o token
    b := balance
    <-sema // libera o token
    return b
}
```

O padrão de *exclusão mútua* é tão útil que é tratado diretamente pelo tipo **Mutex** do pacote **sync**. Seu método **Lock** adquire o token (chamado de *lock* ou trava) e seu método **Unlock** o libera:

gopl.io/ch9/bank3

```
import "sync"

var (
    mu      sync.Mutex // protege balance
    balance int
)
```

```

func Deposit(amount int) {
    mu.Lock()
    balance = balance + amount
    mu.Unlock()
}

func Balance() int {
    mu.Lock()
    b := balance
    mu.Unlock()
    return b
}

```

Sempre que uma gorrotina acessa as variáveis do banco (apenas **balance**, nesse caso), ela deve chamar o método **Lock** do mutex para adquirir uma trava exclusiva. Se outra gorrotina tiver adquirido a trava, essa operação ficará bloqueada até a outra gorrotina chamar **Unlock** e a trava tornar-se disponível de novo. O mutex *protege* (guards) as variáveis compartilhadas. Por convenção, as variáveis protegidas por um mutex são declaradas imediatamente após a declaração do próprio mutex. Se você fizer algo diferente disso, não se esqueça de documentar.

A região de código entre **Lock** e **Unlock**, em que uma gorrotina é livre para ler e modificar as variáveis compartilhadas, é chamada de *seção crítica* (critical section). A chamada do dono da trava a **Unlock** ocorre antes de qualquer outra gorrotina poder adquirir a trava para si. É essencial que a gorrotina libere a trava depois de concluir sua tarefa, em todos os caminhos da função, incluindo os caminhos de erro.

O programa anterior do banco exemplifica um padrão comum de concorrência. Um conjunto de funções exportadas encapsula uma ou mais variáveis, de modo que a única maneira de acessá-las é por meio dessas funções (ou métodos, para as variáveis de um objeto). Cada função adquire uma trava mutex no início e a libera no final, garantindo assim que as variáveis compartilhadas não sejam acessadas de forma concorrente. Esse arranjo de funções, trava mutex e variáveis chama-se *monitor*. (Esse uso antigo da palavra “monitor” inspirou o termo “gorrotina monitora”. Ambos os usos compartilham o sentido de um administrador que garante que as variáveis sejam acessadas

sequencialmente.)

Como as seções críticas das funções **Deposit** e **Balance** são bem pequenas – uma única linha, sem ramificações –, chamar **Unlock** no final é simples. Em seções críticas mais complexas, em especial naquelas em que devemos tratar erros retornando com antecedência, pode ser difícil dizer se as chamadas a **Lock** e a **Unlock** estão rigorosamente pareadas em todos os caminhos. A instrução **defer** de Go nos ajuda: ao adiar uma chamada a **Unlock**, a seção crítica implicitamente estende-se até o final da função atual, deixando-nos livres da obrigação de lembrar de inserir chamadas a **Unlock** em um ou mais lugares distantes da chamada a **Lock**.

```
func Balance() int {  
    mu.Lock()  
    defer mu.Unlock()  
    return balance  
}
```

No exemplo anterior, **Unlock** executa *após* a instrução de retorno ter lido o valor de **balance**, portanto, a função **Balance** é segura para concorrência. Como bônus, não precisamos mais da variável local **b**.

Além disso, um **Unlock** adiar executará mesmo se a seção crítica gerar pânico, o que pode ser importante em programas que usam **recover** (seção 5.10). Um **defer** é um pouco mais custoso que uma chamada explícita a **Unlock**, mas não o suficiente para justificar um código menos claro. Como é frequente em programas concorrentes, favoreça a clareza e resista a otimizações prematuras. Onde for possível, use **defer** e deixe que as seções críticas se estendam até o final de uma função.

Considere a função **Withdraw** a seguir. Se houver sucesso, ela reduz o saldo do valor especificado e devolve **true**. Porém, se a conta não tiver fundos suficientes para a transação, **Withdraw** restaura o saldo e devolve **false**.

```
// NOTA: não é atômico!  
func Withdraw(amount int) bool {  
    Deposit(-amount)  
    if Balance() < 0 {  
        Deposit(amount)  
    }
```

```

        return false // fundo insuficiente
    }
    return true
}

```

No final, essa função fornecerá o resultado correto, mas ela tem um efeito colateral desagradável. Quando houver uma tentativa de saque excessivo, o saldo assume um valor menor que zero momentaneamente. Isso pode fazer um saque concorrente de um valor modesto ser esporadicamente rejeitado. Portanto, se Bob tentar comprar um carro esportivo, Alice não poderá pagar pelo seu café da manhã. O problema é que **Withdraw** não é *atômico*: ele é constituído de uma sequência de três operações diferentes, em que cada uma adquire e, então, libera a trava mutex, mas nenhuma trava a sequência toda.

O ideal é que **Withdraw** adquira a trava mutex uma vez para toda a operação. No entanto, esta tentativa não funcionará:

```

// NOTA: incorreto!
func Withdraw(amount int) bool {
    mu.Lock()
    defer mu.Unlock()
    Deposit(-amount)
    if Balance() < 0 {
        Deposit(amount)
        return false // fundo insuficiente
    }
    return true
}

```

Deposit tenta adquirir a trava mutex uma segunda vez chamando **mu.Lock()**, porém, como as travas mutex não são *reentrantes* – não é possível travar um mutex que já esteja travado –, isso resulta em um deadlock, em que ninguém consegue prosseguir, e **Withdraw** permanece bloqueado indefinidamente.

Há um bom motivo para os mutexes em Go não serem reentrantes. O propósito de um mutex é garantir que determinadas invariantes das variáveis compartilhadas sejam preservadas em pontos críticos durante a execução do programa. Uma das invariantes é “nenhuma gorotina está

acessando as variáveis compartilhadas”, mas pode haver invariantes adicionais específicas às estruturas de dados que o mutex protege. Quando uma gorrotina adquire uma trava mutex, ela pode supor que as invariantes são mantidas. Enquanto está de posse da trava, ela pode atualizar as variáveis compartilhadas de modo que as invariantes sejam temporariamente violadas. No entanto, quando a trava é liberada, é preciso garantir que a ordem seja restabelecida e que as invariantes sejam outra vez preservadas. Embora um mutex reentrante garantisse que nenhuma outra gorrotina acessaria as variáveis compartilhadas, ele não poderia proteger as invariantes adicionais dessas variáveis.

Uma solução comum é dividir uma função como **Deposit** em duas: uma função não exportada, **deposit**, que pressupõe que a trava já está adquirida e que faz o verdadeiro trabalho, e uma função exportada **Deposit** que adquire a trava antes de chamar **deposit**. Podemos então expressar **Withdraw** em função de **deposit**, assim:

```
func Withdraw(amount int) bool {
    mu.Lock()
    defer mu.Unlock()
    deposit(-amount)
    if balance < 0 {
        deposit(amount)
        return false // fundo insuficiente
    }
    return true
}

func Deposit(amount int) {
    mu.Lock()
    defer mu.Unlock()
    deposit(amount)
}

func Balance() int {
    mu.Lock()
    defer mu.Unlock()
    return balance
}

// Esta função exige que a trava tenha sido adquirida.
func deposit(amount int) { balance += amount }
```

É claro que a função **deposit** exibida aqui é tão trivial que uma função **Withdraw** realista não se daria o trabalho de chamá-la, mas ela ilustra o princípio.

Ao reduzir interações inesperadas em um programa, o encapsulamento (seção 6.6) ajuda a preservar as invariantes das estruturas de dados. Pelo mesmo motivo, o encapsulamento também nos ajuda a preservar as invariantes da concorrência. Quando usar um mutex, certifique-se de que tanto ele quanto as variáveis que ele guarda não sejam exportados, independentemente de serem variáveis no nível de pacote ou campos de uma estrutura.

9.3 Mutexes de leitura/escrita: `sync.RWMutex`

Em uma crise de ansiedade após ver seu depósito de cem dólares sumir sem deixar vestígios, Bob escreve um programa para verificar seu saldo bancário centenas de vezes por segundo. Ele o executa em casa, no trabalho e em seu telefone celular. O banco percebe que o aumento no tráfego está atrasando os depósitos e saques, pois todas as requisições a **Balance** executam em sequência, adquirindo a trava com exclusividade e evitando temporariamente que outras gorrotinas executem.

Como a função **Balance** só precisa *ler* o estado da variável, de fato, seria seguro que várias chamadas a **Balance** executassem concorrentemente, desde que nenhuma chamada a **Deposit** ou a **Withdraw** estejam executando. Nesse cenário, precisamos de um tipo especial de trava que permita que operações somente de leitura (read-only) prossigam em paralelo umas com as outras, mas operações de escrita tenham acesso totalmente exclusivo. Essa trava é conhecido como *trava para múltiplas leituras e uma escrita* (*multiple readers, single writer lock*) e, em Go, é oferecido por `sync.RWMutex`:

```
var mu sync.RWMutex
var balance int

func Balance() int {
    mu.RLock() // lock para leituras
    defer mu.RUnlock()
```

```
    return balance  
}
```

A função **Balance** agora chama os métodos **RLock** e **RUnlock** para adquirir e liberar uma trava para *leitura* ou *compartilhado*. A função **Deposit**, que permaneceu inalterada, chama os métodos **mu.Lock** e **mu.Unlock** para adquirir e liberar uma trava para *escrita* ou *exclusivo*.

Após essa alteração, a maior parte das solicitações de **Balance** feitas por Bob executa em paralelo e elas terminam mais rapidamente. A trava está disponível por mais tempo e requisições de **Deposit** podem prosseguir com mais agilidade.

RLock pode ser usado somente se não houver escritas em variáveis compartilhadas na seção crítica. Em geral, não devemos supor que funções ou métodos somente de leitura *logicamente* não atualizem também algumas variáveis. Por exemplo, um método que pareça ser um simples método de acesso também pode incrementar um contador de uso interno ou atualizar um cache de modo que chamadas repetidas sejam mais rápidas. Na dúvida, use um **Lock** exclusivo.

Só vale a pena usar um **RWMutex** quando a maior parte das gorrotinas que adquire a trava são de leitura e a trava está em *disputa* (in contention), isto é, as gorrotinas geralmente devem esperar para adquiri-la. Um **RWMutex** exige uma administração interna mais complexa, tornando-o mais lento que um mutex comum para locks sem contenção.

9.4 Sincronização de memória

Você pode estar se perguntando por que o método **Balance** precisa de exclusão mútua, seja baseada em canal, seja em mutex. Afinal de contas, de modo diferente de **Deposit**, ele é constituído apenas de uma única operação, portanto, não há perigo de outra gorrotina executar “no meio” dela. Há dois motivos para precisarmos de um mutex. O primeiro é que é igualmente importante que **Balance** não execute no meio de outra operação como **Withdraw**. O segundo motivo (e mais sutil) é que a sincronização não está relacionada somente com a ordem de execução de

várias gorrotinas, mas ela também afeta a memória.

Em um computador moderno pode haver dezenas de processadores, cada um com seu próprio cache local da memória principal. Por questões de eficiência, escritas na memória são colocadas em um buffer em cada processador e descarregadas para a memória principal somente quando for necessário. Elas podem até mesmo ser efetivadas na memória principal em uma ordem diferente daquela em que foram escritas pela gorrotina de escrita. Primitivas de sincronização como comunicações por canal e operações de mutex fazem o processador descarregar e efetivar todas as escritas acumuladas de modo que se garanta que os efeitos da execução de uma gorrotina até esse ponto sejam visíveis às gorrotinas executando em outros processadores.

Considere as saídas possíveis do trecho de código a seguir:

```
var x, y int
go func() {
    x = 1                // A1
    fmt.Print("y:", y, " ") // A2
}()
go func() {
    y = 1                // B1
    fmt.Print("x:", x, " ") // B2
}()
```

Como essas duas gorrotinas são concorrentes e acessam variáveis compartilhadas sem exclusão mútua, há uma concorrência de dados, portanto, não devemos ficar surpresos com o fato de o programa não ser determinístico. Podemos esperar que ele exiba qualquer um dos quatro resultados a seguir, que correspondem a entrelaçamentos intuitivos das instruções identificadas no programa:

```
y:0 x:1
x:0 y:1
x:1 y:1
y:1 x:1
```

A quarta linha poderia ser explicada pela sequência **A1,B1,A2,B2** ou por **B1,A1,A2,B2**, por exemplo. Entretanto, estes dois resultados podem causar surpresa:

```
x:0 y:0  
y:0 x:0
```

mas, dependendo do compilador, da CPU e de muitos outros fatores, eles também podem ocorrer. Que possível entrelaçamento das quatro instruções poderia explicá-los?

Em uma única gorrotina, garante-se que os efeitos de cada instrução ocorram na ordem de execução; as gorrotinas são *sequencialmente consistentes*. Porém, na ausência de uma sincronização explícita usando um canal ou um mutex, não há garantias de que os eventos sejam vistos na mesma ordem por todas as gorrotinas. Embora a gorrotina A deva observar o efeito da escrita `x = 1` antes de ler o valor de `y`, ela não necessariamente observa a escrita em `y` feita pela gorrotina B, portanto, A pode exibir um valor *ultrapassado* de `y`.

É tentador procurar entender a concorrência como se ela correspondesse a *algum* entrelaçamento de instruções de cada gorrotina, porém, como o exemplo anterior mostra, não é assim que um compilador ou uma CPU modernos funcionam. Como a atribuição e o `Print` referem-se a variáveis diferentes, um compilador pode concluir que a ordem das duas instruções não pode afetar o resultado, e trocá-las. Se as duas gorrotinas executarem em CPUs diferentes, cada um com seu próprio cache, as escritas feitas por uma gorrotina não serão visíveis ao `Print` da outra gorrotina até os caches estarem sincronizados com a memória principal.

Todos esses problemas de concorrência podem ser evitados com o uso consistente de padrões simples e definidos. Sempre que possível, confine variáveis em uma única gorrotina; para todas as demais variáveis, use exclusão mútua.

9.5 Inicialização lazy: `sync.Once`

É uma boa prática adiar um passo custoso de inicialização até o momento em que ele for necessário. Inicializar uma variável com antecedência aumenta a latência de inicialização de um programa e é desnecessário se a execução nem sempre alcançar a parte do programa que usa essa variável.

Vamos retomar a variável **icons** que vimos antes neste capítulo:

```
var icons map[string]image.Image
```

Esta versão de **Icon** usa *inicialização preguiçosa* (lazy initialization):

```
func loadIcons() {
    icons = map[string]image.Image{
        "spades.png":  loadIcon("spades.png"),
        "hearts.png":  loadIcon("hearts.png"),
        "diamonds.png": loadIcon("diamonds.png"),
        "clubs.png":   loadIcon("clubs.png"),
    }
}

// NOTA: não é seguro para concorrência!
func Icon(name string) image.Image {
    if icons == nil {
        loadIcons() // inicialização feita uma só vez
    }
    return icons[name]
}
```

Para uma variável acessada somente por uma única gorrotina, podemos usar o padrão anterior, porém esse padrão não é seguro se **Icon** for chamada de forma concorrente. Assim como a função **Deposit** original do banco, **Icon** é constituída de vários passos: ela testa se **icons** é **nil** e, então, carrega os ícones; em seguida, ela atualiza **icons** com um valor diferente de **nil**. A intuição pode sugerir que o pior resultado possível da condição de corrida anterior é que a função **loadIcons** seja chamada várias vezes. Enquanto a primeira gorrotina está ocupada carregando os ícones, outra gorrotina que entrar em **Icon** encontraria a variável ainda igual a **nil** e também chamaria **loadIcons**.

Contudo essa intuição também está incorreta. (Esperamos que, a essa altura, você esteja desenvolvendo uma nova intuição sobre concorrência, isto é, que intuições sobre concorrência não são confiáveis!) Lembre-se da discussão sobre memória da seção 9.4. Na ausência de uma sincronização explícita, o compilador e a CPU estão livres para reordenar os acessos à memória de várias maneiras, desde que o comportamento de cada gorrotina seja sequencialmente consistente. Uma possível reordenação das

instruções de **loadIcons** é apresentada a seguir. Ela armazena o mapa vazio na variável **icons** antes de preenchê-la:

```
func loadIcons() {
    icons = make(map[string]image.Image)
    icons["spades.png"] = loadIcon("spades.png")
    icons["hearts.png"] = loadIcon("hearts.png")
    icons["diamonds.png"] = loadIcon("diamonds.png")
    icons["clubs.png"] = loadIcon("clubs.png")
}
```

Consequentemente, uma gorrotina que se depare com **icons** diferente de nil não pode supor que a inicialização da variável esteja concluída.

A maneira mais simples e correta de garantir que todas as gorrotinas observem os efeitos de **loadIcons** é sincronizá-las usando um mutex:

```
var mu sync.Mutex // guarda icons
var icons map[string]image.Image

// Seguro para concorrência.
func Icon(name string) image.Image {
    mu.Lock()
    defer mu.Unlock()
    if icons == nil {
        loadIcons()
    }
    return icons[name]
}
```

No entanto, o custo de impor um acesso mutuamente exclusivo a **icons** é que duas gorrotinas não podem acessar a variável concorrentemente, mesmo depois que esta tiver sido inicializada com segurança e que jamais seja modificada novamente. Isso sugere uma trava para várias leituras:

```
var mu sync.RWMutex // guarda icons
var icons map[string]image.Image

// Seguro para concorrência.
func Icon(name string) image.Image {
    mu.RLock()
    if icons != nil {
        icon := icons[name]
        mu.RUnlock()
        return icon
    }
}
```

```

    }
    mu.RUnlock()

    // adquire uma trava exclusiva
    mu.Lock()
    if icons == nil { // NOTA: devemos verificar novamente se é nil
        loadIcons()
    }
    icon := icons[name]
    mu.Unlock()
    return icon
}

```

Agora temos duas seções críticas. A gorrotina inicialmente adquire uma trava para leitura, consulta o mapa e então libera a trava. Se uma entrada for encontrada (o caso comum), ela é devolvida. Se nenhuma entrada foi encontrada, a gorrotina adquire uma trava para escrita. Não há como transformar uma trava compartilhada em uma trava exclusiva, sem antes liberar a trava compartilhada, portanto, precisamos verificar novamente a variável **icons** no caso de outra gorrotina já tê-la inicializado nesse ínterim.

O padrão anterior suporta mais concorrência, mas é complexo e, portanto, suscetível a erros. Felizmente o pacote **sync** oferece uma solução especializada para o problema da inicialização única: **sync.Once**. Conceitualmente, um **Once** é constituído de um mutex e de uma variável booleana que registra se a inicialização já ocorreu; o mutex guarda tanto o booleano quando as estruturas de dados do cliente. Seu único método **Do** aceita a função de inicialização como argumento. Vamos usar **Once** para simplificar a função **Icon**:

```

var loadIconsOnce sync.Once
var icons map[string]image.Image

// Seguro para concorrência.
func Icon(name string) image.Image {
    loadIconsOnce.Do(loadIcons)
    return icons[name]
}

```

Cada chamada a **Do(loadIcons)** trava o mutex e verifica a variável booleana. Na primeira chamada, em que a variável é falsa, **Do** chama

`loadIcons` e define a variável como verdadeira. Chamadas subseqüente não fazem nada, porém a sincronização com o mutex garante que os efeitos de `loadIcons` na memória (especificamente, `icons`) tornem-se visíveis a todas as gorrotinas. Ao usar `sync.Once` dessa maneira, podemos evitar o compartilhamento de variáveis com outras gorrotinas até que elas tenham sido apropriadamente construídas.

Exercício 9.2: Reescreva o exemplo com `PopCount` da seção 2.6.2 para que ele inicialize a tabela de busca usando `sync.Once` na primeira vez em que ela for necessária. (Falando de modo realista, o custo da sincronização seria proibitivo para uma função pequena e altamente otimizada como `PopCount`.)

9.6 O detector de concorrência

Mesmo com o máximo de cuidados possível, é fácil demais cometer erros quando se trata de concorrência. Felizmente, o runtime de Go e o conjunto de ferramentas são equipados com uma ferramenta de análise sofisticada, dinâmica e fácil de usar: o *detector de concorrência* (race detector).

Basta acrescentar a flag `-race` ao seu comando `go build`, `go run` ou `go test`. Isso faz o compilador criar uma versão modificada de sua aplicação ou de seu teste com uma instrumentação adicional que registra todos os acessos a variáveis compartilhadas que ocorreram durante a execução, juntamente com a identidade da gorrotina que leu ou escreveu na variável. Além disso, o programa modificado registra todos os eventos de sincronização, como as instruções `go`, as operações em canais e chamadas a `(*sync.Mutex).Lock`, `(*sync.WaitGroup).Wait`, e assim por diante. [O conjunto completo de eventos de sincronização está especificado no documento *The Go Memory Model* (O modelo de memória de Go) que acompanha a especificação da linguagem.]

O detector de concorrência analisa esse fluxo de eventos, procurando casos em que uma gorrotina lê ou escreve em uma variável compartilhada que foi mais recentemente escrita por uma gorrotina diferente, sem uma

operação de sincronização interveniente. Isso indica um acesso concorrente à variável compartilhada e, portanto, uma concorrência de dados. A ferramenta exibe um relatório que inclui a identidade da variável e as pilhas das chamadas de função ativas na gorrotina de leitura e na gorrotina de escrita. Geralmente, isso é suficiente para identificar o problema. A seção 9.7 contém um exemplo do detector de concorrência em ação.

O detector de concorrência informa todas as concorrências de dados que foram realmente executadas. No entanto, ele pode detectar apenas as condições de concorrência que ocorrem durante uma execução; não é capaz de provar que nenhuma ocorrerá. Para obter os melhores resultados possível, certifique-se de que seus testes exercitem seus pacotes usando concorrência.

Por causa da administração extra, um programa criado com detecção de concorrência precisa de mais tempo e de mais memória para executar, porém o overhead é tolerável, mesmo para muitas tarefas em ambiente de produção. Para condições de concorrência que não ocorram com muita frequência, deixar o detector de concorrência fazer seu trabalho pode economizar horas ou dias de depuração.

9.7 Exemplo: Cache concorrente não bloqueante

Nesta seção, criaremos um *cache concorrente não bloqueante*, isto é, uma abstração que resolve um problema que surge com frequência em programas concorrentes do mundo real, mas que não é bem tratado por bibliotecas existentes. É o problema da *memoização* (memoizing) de uma função, ou seja, fazer caching do resultado de uma função para que ela só precise ser processada uma vez. Nossa solução será segura para concorrência e evitará as disputas associadas a designs baseados em um única trava para todo o cache.

Usaremos a função `httpGetBody` a seguir como exemplo do tipo de função que queremos utilizar para memoizar. Ela faz uma requisição HTTP GET e lê o corpo da resposta. Chamadas a essa função são

relativamente custosas, portanto, gostaríamos de evitar repeti-las desnecessariamente.

```
func httpGetBody(url string) (interface{}, error) {
    resp, err := http.Get(url)
    if err != nil {
        return nil, err
    }
    defer resp.Body.Close()
    return ioutil.ReadAll(resp.Body)
}
```

A última linha esconde uma pequena sutileza. **ReadAll** devolve dois resultados, um `[]byte` e um **error**, mas como eles podem ser atribuídos aos tipos de resultado declarados de **httpGetBody** – `interface{}` e **error**, respectivamente – podemos simplesmente devolver o resultado da chamada. Escolhemos esse tipo de retorno para **httpGetBody** para que ele esteja de acordo com o tipo de funções para o qual nosso cache foi projetado para memoizar.

Eis a primeira versão preliminar do cache:

gopl.io/ch9/memo1

```
// O pacote memo oferece uma memoização
// segura para concorrência de uma função do tipo Func.
package memo

// Um Memo faz cache dos resultados da chamada a uma Func.
type Memo struct {
    f      Func
    cache map[string]result
}

// Func é o tipo da função para memoizar.
type Func func(key string) (interface{}, error)
type result struct {
    value interface{}
    err error
}

func New(f Func) *Memo {
    return &Memo{f: f, cache: make(map[string]result)}
}

// NOTA: não é seguro para concorrência!
```

```
func (memo *Memo) Get(key string) (interface{}, error) {
    res, ok := memo.cache[key]
    if !ok {
        res.value, res.err = memo.f(key)
        memo.cache[key] = res
    }
    return res.value, res.err
}
```

Uma instância de **Memo** armazena a função **f** do tipo **Func** a ser memoizada e o cache, que é um mapeamento de strings para **results**. Cada **result** é simplesmente o par de resultados devolvidos por uma chamada a **f** – um valor e um erro. Mostraremos diversas variações de **Memo** à medida que o design evoluir, mas todas compartilharão esses aspectos básicos.

Apresentamos um exemplo de como usar **Memo** a seguir. Para cada elemento em um fluxo de URLs de entrada, chamamos **Get**, fazendo log da latência da chamada e da quantidade de dados que ela devolve:

```
m := memo.New(httpGetBody)
for url := range incomingURLs() {
    start := time.Now()
    value, err := m.Get(url)
    if err != nil {
        log.Print(err)
    }
    fmt.Printf("%s, %s, %d bytes\n",
        url, time.Since(start), len(value.([]byte)))
}
```

Podemos usar o pacote **testing** (que será o assunto do capítulo 11) para investigar sistematicamente o efeito da memoização. A partir do resultado do teste a seguir, vemos que o fluxo de URLs contém duplicatas e que, embora a primeira chamada a **(*Memo).Get** para cada URL demore centenas de milissegundos, a segunda requisição devolve a mesma quantidade de dados em menos de um milissegundo.

```
$ go test -v gopl.io/ch9/memo1
=== RUN    Test
https://golang.org, 175.026418ms, 7537 bytes
https://godoc.org, 172.686825ms, 6878 bytes
```

```
https://play.golang.org, 115.762377ms, 5767 bytes
http://gopl.io, 749.887242ms, 2856 bytes
https://golang.org, 721ns, 7537 bytes
https://godoc.org, 152ns, 6878 bytes
https://play.golang.org, 205ns, 5767 bytes
http://gopl.io, 326ns, 2856 bytes
--- PASS: Test (1.21s)
PASS
ok  gopl.io/ch9/memo1    1.257s
```

Esse teste executa todas as chamadas a **Get** sequencialmente.

Como as requisições HTTP são uma ótima oportunidade para paralelismo, vamos alterar o teste de modo que ele faça todas as requisições de forma concorrente. O teste usa um **sync.WaitGroup** para esperar até que a última requisição esteja concluída antes de retornar.

```
m := memo.New(httpGetBody)
var n sync.WaitGroup
for url := range incomingURLs() {
    n.Add(1)
    go func(url string) {
        start := time.Now()
        value, err := m.Get(url)
        if err != nil {
            log.Print(err)
        }
        fmt.Printf("%s, %s, %d bytes\n",
            url, time.Since(start), len(value.([]byte)))
        n.Done()
    }(url)
}
n.Wait()
```

O teste executa bem mais rápido, mas, infelizmente, é pouco provável que funcione de forma correta o tempo todo. Podemos perceber ausências inesperadas de cache ou acessos que encontram dados no cache, mas devolvem valores incorretos, ou até mesmo causam falhas.

Pior ainda, é provável que ele funcione corretamente *parte* do tempo, portanto, podemos nem perceber que há um problema. Porém, se executarmos o teste com a flag **-race**, o detector de concorrência (seção 96) em geral exibirá um relatório como este:

```

$ go test -run=TestConcurrent -race -v gopl.io/ch9/memo1
=== RUN    TestConcurrent
...
WARNING: DATA RACE
Write by goroutine 36:
    runtime.mapassign1()
        ~/go/src/runtime/hashtable.go:411 +0x0
    gopl.io/ch9/memo1.(*Memo).Get()
        ~/gobook2/src/gopl.io/ch9/memo1/memo.go:32 +0x205
    ...
Previous write by goroutine 35:
    runtime.mapassign1()
        ~/go/src/runtime/hashtable.go:411 +0x0
    gopl.io/ch9/memo1.(*Memo).Get()
        ~/gobook2/src/gopl.io/ch9/memo1/memo.go:32 +0x205
    ...
Found 1 data race(s)
FAIL    gopl.io/ch9/memo1    2.393s

```

A referência a **memo.go:32** informa que duas gorrotinas atualizaram o mapa **cache** sem nenhuma sincronização interferindo. **Get** não é segura para concorrência: ela tem uma concorrência de dados.

```

28 func (memo *Memo) Get(key string) (interface{}, error) {
29     res, ok := memo.cache[key]
30     if !ok {
31         res.value, res.err = memo.f(key)
32         memo.cache[key] = res
33     }
34     return res.value, res.err
35 }

```

A maneira mais simples de deixar o cache seguro para concorrência é usar sincronização baseada em monitor. Tudo o que precisamos fazer é acrescentar um mutex em **Memo**, adquirir a trava mutex no início de **Get** e liberá-la antes de **Get** retornar, de modo que as duas operações em **cache** ocorram dentro da seção crítica:

gopl.io/ch9/memo2

```

type Memo struct {
    f      Func
    mu     sync.Mutex // guarda cache
    cache  map[string]result

```

```

}
// Get é segura para concorrência.
func (memo *Memo) Get(key string) (value interface{}, err error) {
    memo.mu.Lock()
    res, ok := memo.cache[key]
    if !ok {
        res.value, res.err = memo.f(key)
        memo.cache[key] = res
    }
    memo.mu.Unlock()
    return res.value, res.err
}

```

Agora o detector de concorrência está silencioso, mesmo quando executamos os testes de forma concorrente. Infelizmente, essa mudança em **Memo** reverte nossos ganhos anteriores de desempenho. Ao estar de posse da trava pela duração de cada chamada a **f**, **Get** serializa todas as operações de E/S que pretendíamos paralelizar. O que precisamos é de um cache *não bloqueante*: um que não serialize as chamadas à função que vai ser memoizada.

Na implementação de **Get** a seguir, a gorrotina que faz a chamada adquire a trava duas vezes: uma para a busca e outra para a atualização, se a busca não devolveu nada. Entre elas, outras gorrotinas estão livres para usar o cache.

gopl.io/ch9/memo3

```

func (memo *Memo) Get(key string) (value interface{}, err error) {
    memo.mu.Lock()
    res, ok := memo.cache[key]
    memo.mu.Unlock()
    if !ok {
        res.value, res.err = memo.f(key)

        // Entre as duas seções críticas, várias gorrotinas
        // podem concorrer para processar f(key) e atualizar o mapa.
        memo.mu.Lock()
        memo.cache[key] = res
        memo.mu.Unlock()
    }
    return res.value, res.err
}

```

```
}
```

O desempenho melhora novamente, mas, agora, percebemos que alguns URLs estão sendo buscados duas vezes. Isso acontece quando duas ou mais gorrotinas chamam **Get** para o mesmo URL quase ao mesmo tempo. Ambas consultam o cache, não encontram nenhum valor ali e então chamam a função lenta **f**. Em seguida, ambas atualizam o mapa com o resultado que obtiveram. Um dos resultados é sobrescrito pelo outro.

O ideal seria evitar esse trabalho redundante. Esse recurso às vezes é chamado de *supressão de duplicações* (duplicate suppression). Na versão de **Memo** a seguir, cada elemento do mapa é um ponteiro para uma estrutura **entry**. Cada **entry** contém o resultado memoizado de um chamado à função **f**, como antes, mas contém também um canal chamado **ready**. Logo após o **result** de **entry** ter sido definido, esse canal será fechado para fazer *broadcast* (seção 8.9) a quaisquer outras gorrotinas, informando que agora é seguro para elas lerem o resultado de **entry**.

gopl.io/ch9/memo4

```
type entry struct {
    res    result
    ready chan struct{} // fechado quando res estiver pronto
}

func New(f Func) *Memo {
    return &Memo{f: f, cache: make(map[string]*entry)}
}

type Memo struct {
    f      Func
    mu     sync.Mutex // guarda cache
    cache map[string]*entry
}

func (memo *Memo) Get(key string) (value interface{}, err error) {
    memo.mu.Lock()
    e := memo.cache[key]
    if e == nil {
        // Essa é a primeira requisição para essa chave.
        // Essa gorrotina torna-se responsável por calcular
        // o valor e fazer broadcast da condição de pronto.
    }
}
```



```

    e = &entry{ready: make(chan struct{})}
    memo.cache[key] = e
    memo.mu.Unlock()

    e.res.value, e.res.err = memo.f(key)

    close(e.ready) // faz broadcast da condição de pronto
} else {
    // Essa é a uma requisição repetida para essa chave.
    memo.mu.Unlock()

    <-e.ready // espera a condição de pronto
}
return e.res.value, e.res.err
}

```

Uma chamada a **Get** agora envolve adquirir a trava mutex que protege o mapa **cache**, buscar no mapa um ponteiro para uma **entry** existente, alocar e inserir uma nova **entry** se nenhuma for encontrada e, então, liberar a trava. Se houver uma **entry** existente, seu valor não estará necessariamente pronto – outra gorrotina poderia ainda estar chamando a função lenta **f**. Portanto, a gorrotina que fizer a chamada deve esperar a condição de “pronto” de **entry** antes de ler o **result** de **entry**. Isso é feito lendo um valor do canal **ready**, pois essa operação fica bloqueada até o canal ser fechado.

Se não houver nenhuma **entry** existente, então, ao inserir uma nova **entry** “não pronta” no mapa, a gorrotina atual torna-se responsável por chamar a função lenta, atualizar **entry** e fazer broadcast do estado de pronto da nova **entry** a qualquer outra gorrotina que possa (então) estar esperando por ela.

Observe que as variáveis **e.res.value** e **e.res.err** em **entry** são compartilhadas entre várias gorrotinas. A gorrotina que cria **entry** define seus valores, e outras gorrotinas os leem depois que a condição “pronto” foi informada via broadcast. Apesar de ser acessada por várias gorrotinas, nenhuma trava mutex é necessária. O fechamento do canal **ready** *acontece antes* de qualquer outra gorrotina receber o evento de broadcast, portanto, a escrita nessas variáveis na primeira gorrotina *acontece antes* de elas serem lidas por gorrotinas subsequentes. Não há concorrência de

dados.

Nosso cache concorrente, que suprime duplicações e é não bloqueante, está completo.

A implementação do **Memo** anterior usa um mutex para guardar uma variável de mapa compartilhada por todas as gorrotinas que chamam **Get**. É interessante comparar esse design com um design alternativo em que a variável de mapa é confinada em uma *gorrotina monitora* à qual quem chama **Get** deve enviar uma mensagem.

As declarações de **Func**, **result** e **entry** permanecem iguais:

```
// Func é o tipo da função para memoizar.
type Func func(key string) (interface{}, error)

// Um result é o resultado de uma chamada a Func.
type result struct {
    value interface{}
    err error
}

type entry struct {
    res result
    ready chan struct{} // fechado quando res estiver pronto
}
```

No entanto, o tipo **Memo** agora é constituído de um canal **requests** por meio do qual quem chama **Get** se comunica com a gorrotina monitora. O tipo do elemento do canal é um **request**. Usando essa estrutura, quem chama **Get** envia à gorrotina monitora tanto a chave, isto é, o argumento para a função memoizada, quanto outro canal, **response**, por meio do qual o resultado deve ser enviado de volta quando estiver disponível. Esse canal transportará apenas um único valor.

gopl.io/ch9/memo5

```
// Um request é uma mensagem que solicita que Func seja aplicada a key.
type request struct {
    key      string
    response chan<- result // o cliente quer um único resultado
}

type Memo struct{ requests chan request }
```

```
// New devolve uma memoização de f. Clientes devem chamar Close
subsequentemente.
func New(f Func) *Memo {
    memo := &Memo{requests: make(chan request)}
    go memo.server(f)
    return memo
}

func (memo *Memo) Get(key string) (interface{}, error) {
    response := make(chan result)
    memo.requests <- request{key, response}
    res := <-response
    return res.value, res.err
}

func (memo *Memo) Close() { close(memo.requests) }
```

O método **Get** anterior cria um canal de resposta, coloca-o na requisição, envia-o à gorrotina monitora e então recebe imediatamente desse canal.

A variável **cache** é confinada na gorrotina monitora, **(*Memo).server**, mostrada a seguir. O monitor lê requisições em um loop até o canal de solicitação ser fechado pelo método **Close**. Para cada requisição, ele consulta o cache, criando e inserindo uma nova **entry** se nenhuma for encontrada.

```
func (memo *Memo) server(f Func) {
    cache := make(map[string]*entry)
    for req := range memo.requests {
        e := cache[req.key]
        if e == nil {
            // Essa é a primeira requisição para essa chave.
            e = &entry{ready: make(chan struct{})}
            cache[req.key] = e
            go e.call(f, req.key) // chama f(key)
        }
        go e.deliver(req.response)
    }
}

func (e *entry) call(f Func, key string) {
    // Avalia a função.
    e.res.value, e.res.err = f(key)
    // Faz broadcast da condição de pronto.
    close(e.ready)
}
```

```

}
func (e *entry) deliver(response chan<- result) {
    // Espera a condição de pronto.
    <-e.ready
    // Envia o resultado ao cliente.
    response <- e.res
}

```

Como na versão baseada em mutex, a primeira requisição para uma dada chave torna-se responsável por chamar a função `f` nessa chave, armazenando o resultado em `entry` e fazendo o broadcast para informar que `entry` está pronto fechando o canal `ready`. Isso é feito por `(*entry).call`.

Uma requisição subsequente para a mesma chave encontra a `entry` existente no mapa, espera que o resultado fique pronto e envia-o por meio do canal de resposta à gorrotina cliente que chamou `Get`. Isso é feito por `(*entry).deliver`. Os métodos `call` e `deliver` devem ser chamados em suas próprias gorrotinas para garantir que a gorrotina monitora não pare de processar novas requisições.

Esse exemplo mostra que é possível criar muitas estruturas concorrentes usando uma de duas abordagens – variáveis compartilhadas e locks ou processos sequenciais comunicantes (communicating sequential processes) – sem uma complexidade excessiva.

Nem sempre é óbvio qual abordagem é preferível em uma dada situação, mas vale a pena saber a relação entre elas. Às vezes, passar de uma abordagem para outra pode deixar seu código mais simples.

Exercício 9.3: Estenda o tipo `Func` e o método `(*Memo).Get` para que quem chama possa oferecer um canal `done` opcional por meio do qual seja possível cancelar a operação (seção 8.9). O resultado de uma chamada cancelada a `Func` não deve ser inserido no cache.

9.8 Gorrotinas e threads

No capítulo anterior, dissemos que a diferença entre gorrotinas e threads do sistema operacional (SO) poderiam ser ignoradas por um tempo.

Embora as diferenças entre elas sejam essencialmente quantitativas, uma diferença quantitativa grande o suficiente torna-se qualitativa, e é isso o que ocorre com gorrotinas e threads. Chegou a hora de diferenciá-las.

9.8.1 Pilhas que podem aumentar

Cada thread de sistema operacional tem um bloco de memória de tamanho fixo (geralmente tão grande quanto 2 MB) para sua *pilha* (stack) – a área de trabalho em que ela salva as variáveis locais de chamadas de função que estão em progresso ou temporariamente suspensas enquanto outra função é chamada. Essa pilha de tamanho fixo é, ao mesmo tempo, muito grande e muito pequena. Uma pilha de 2 MB seria um enorme desperdício de memória para uma gorrotina pequena, como aquela que simplesmente espera um `WaitGroup` e então fecha um canal. Não é incomum que um programa Go crie centenas de milhares de gorrotinas ao mesmo tempo, o que seria impossível com pilhas desse tamanho. Apesar de seu tamanho, pilhas de tamanho fixo nem sempre são grandes o suficiente para as funções mais complexas e profundamente recursivas. Alterar o tamanho fixo pode melhorar a eficiência quanto ao espaço e permitir que mais threads sejam criadas, ou pode permitir funções recursivas de maior profundidade, mas não pode fazer as duas coisas.

Em comparação, uma gorrotina começa a vida com uma pilha pequena, normalmente de 2 KB. A pilha de uma gorrotina, assim como a de uma thread do sistema operacional, armazena as variáveis locais de chamadas de funções ativas e suspensas, porém, de modo diferente das threads do sistema operacional, a pilha de uma gorrotina não é fixa; ela cresce e diminui conforme a necessidade. O limite de tamanho para uma pilha de gorrotina pode ser tão grande quanto 1 GB, que é uma ordem de grandeza maior que a pilha de tamanho fixo típica de uma thread, embora, é claro, poucas gorrotinas usem todo esse espaço.

Exercício 9.4: Crie um pipeline que conecte um número arbitrário de gorrotinas com canais. Qual é o número máximo de estágios de pipeline que você pode criar sem ficar sem memória? Quanto tempo um valor

demora para passar por todo o pipeline?

9.8.2 Escalonamento de gorrotinas

Threads de sistema operacional são escalonadas pelo kernel do sistema. A intervalos de alguns milissegundos, um timer do hardware interrompe o processador, o que faz uma função do kernel chamada *escalonador* (scheduler) ser chamada. Essa função suspende a thread que está executando no momento e salva seus registradores na memória, percorre a lista de threads e decide qual delas deve executar a seguir, restaura os registradores dessa thread da memória e, então, retoma a sua execução. Como as threads do sistema operacional são escalonadas pelo kernel, passar o controle de uma thread para outra exige uma *mudança de contexto* (context switch) completa, isto é, salvar o estado de uma thread de usuário na memória, restaurar o estado de outra e atualizar as estruturas de dados do escalonador. Essa operação é lenta por causa de sua localidade precária³ e do número de acessos à memória exigido e, historicamente, só piorou à medida que o número de ciclos de CPU necessários para acessar a memória tem aumentado.

O runtime de Go contém seu próprio escalonador que usa uma técnica conhecida como *escalonamento m:n* (m:n scheduling) porque ela multiplexa (ou escalona) m gorrotinas em n threads do sistema operacional. A tarefa do escalonador de Go é análoga à tarefa do escalonador do kernel, mas o escalonador de Go está preocupado somente com as gorrotinas de um único programa Go.

Diferente do escalonador de threads do sistema operacional, o escalonador de Go não é chamado periodicamente por um timer do hardware, mas é chamado de forma implícita por determinadas construções da linguagem Go. Por exemplo, quando uma gorrotina chama `time.Sleep` ou fica bloqueada em um canal ou em uma operação de mutex, o escalonador coloca-a para dormir e executa outra gorrotina até chegar o momento de acordar a primeira. Como ele não precisa fazer uma mudança para o contexto do kernel, reescalonar uma gorrotina é muito

mais simples do que reescalonar uma thread.

Exercício 9.5: Escreva um programa com duas gorrotinas que enviam mensagens de um lado para outro por meio de dois canais sem buffer, como se fosse um pingue-pongue. Quantas comunicações por segundo o programa é capaz de sustentar?

9.8.3 GOMAXPROCS

O escalonador de Go usa um parâmetro chamado **GOMAXPROCS** para determinar quantas threads de sistema operacional podem estar ativas executando código Go simultaneamente. Seu valor default é o número de CPUs do computador, portanto, em um computador com oito CPUs, o escalonador escalonará código Go em até oito threads do sistema operacional ao mesmo tempo. (**GOMAXPROCS** é o n no escalonamento $m:n$.) Gorrotinas que estão dormindo ou que estejam bloqueadas em uma comunicação não precisam de uma thread. Gorrotinas bloqueadas em E/S ou em outras chamadas de sistema, ou que estejam chamando funções que não estão implementadas em Go, precisam de uma thread do sistema operacional, mas **GOMAXPROCS** não precisa levá-las em consideração.

Você pode controlar explicitamente esse parâmetro usando a variável de ambiente **GOMAXPROCS** ou a função **runtime.GOMAXPROCS**. Podemos ver o efeito de **GOMAXPROCS** neste pequeno programa, que exibe um fluxo interminável de zeros e uns:

```
for {  
    go fmt.Print(0)  
    fmt.Print(1)  
}  
  
$ GOMAXPROCS=1 go run hacker-cliché.go  
111111111111111111111111110000000000000000000000011111...  
  
$ GOMAXPROCS=2 go run hacker-cliché.go  
0101010101010101010101011001100101011010010100110...
```

Na primeira execução, no máximo uma gorrotina foi executada em um determinado instante. Inicialmente, foi a gorrotina principal, que exibe uns. Após um período de tempo, o escalonador de Go colocou-a para

dormir e acordou a gorrotina que exhibe zeros, dando-lhe uma oportunidade para executar na thread do sistema operacional. Na segunda execução, havia duas threads de sistema operacional disponíveis, portanto, ambas as gorrotinas executaram em simultâneo, exibindo dígitos aproximadamente na mesma taxa. Devemos enfatizar que muitos fatores estão envolvidos no escalonamento de gorrotinas, e o runtime está sempre evoluindo, portanto, seus resultados poderão ser diferentes dos apresentados anteriormente.

Exercício 9.6: Avalie como o desempenho de um programa paralelo limitado por processamento (compute-bound) – veja o exercício 8.5 – varia de acordo com GOMAXPROCS. Qual é o valor ideal em seu computador? Quantas CPUs seu computador tem?

9.8.4 Gorrotinas não têm identidade

Na maioria dos sistemas operacionais e linguagens de programação que tratam multithreading, a thread atual tem uma identidade distinta que pode ser facilmente obtida como um valor comum, em geral um inteiro ou um ponteiro. Isso facilita criar uma abstração chamada *armazenamento local da thread* (thread-local storage) que na essência é um mapa global em que a identidade da thread é a chave, de modo que cada thread possa armazenar e recuperar valores independentemente de outras threads.

As gorrotinas não têm nenhuma noção de identidade que seja acessível ao programador. Isso foi projetado para ser assim, pois há uma tendência a abusos no armazenamento local de thread. Por exemplo, em um servidor web implementado em uma linguagem com armazenamento local de thread, é comum que muitas funções encontrem informações sobre a requisição HTTP para a qual elas estão trabalhando no momento consultando essa área de armazenamento. Contudo, assim como programas que contam excessivamente com variáveis globais, isso pode resultar em uma “ação à distância” prejudicial, em que o comportamento de uma função não é determinado somente por seus argumentos, mas

pela identidade da thread em que ela é executada. Consequentemente, se a identidade da thread precisar mudar – por exemplo, algumas threads de trabalho são convocadas a ajudar –, a função passa como que por mágica a se comportar mal.

Go incentiva um estilo mais simples de programação, em que parâmetros que afetem o comportamento de uma função devem ser explícitos. Isso não só deixa os programas mais legíveis, como também permite atribuir livremente subtarefas de uma dada função a várias gorrotinas diferentes sem nos preocuparmos com suas identidades.

Agora você já viu todos os recursos da linguagem Go necessários para escrever programas. Nos dois próximos capítulos, daremos um passo para trás e veremos algumas práticas e ferramentas que dão suporte à programação em geral: como estruturar um projeto como um conjunto de pacotes e como obter, compilar, testar, fazer benchmark e profiling, documentar e compartilhar esses pacotes.

¹ N.T.: Um *livelock* é semelhante a um *deadlock*, exceto que os estados dos processos envolvidos no *livelock* mudam constantemente um em relação ao outro, e nenhum deles progride. (Baseado em <https://en.wikipedia.org/wiki/Deadlock#Livelock>).

² N.T.: “Em programação concorrente, ocorre inanição quando um processo nunca é executado (‘morre de fome’), pois processos de prioridade maior sempre o impedem de ser executado.” (Fonte: [https://pt.wikipedia.org/wiki/Inanição_\(computação\)](https://pt.wikipedia.org/wiki/Inanição_(computação)))

³ Nota do Revisor da Tradução: a expressão original era “poor locality”. Aqui, “locality” refere-se à proximidade dos valores necessários para uma operação. Uma boa localidade é, por exemplo, todos os valores no cache L1 da CPU. Uma localidade precária ou desfavorável significa que os valores necessários a uma operação estão armazenados em locais distantes, exigindo mais ciclos de CPU para sua recuperação. Veja: https://pt.wikipedia.org/wiki/Localidade_de_referência.

10

Pacotes e a ferramenta go

Um programa de tamanho modesto atualmente pode conter dez mil funções. Apesar disso, seu autor precisa pensar em apenas algumas delas e fazer o design de um número menor ainda, pois a grande maioria foi escrita por outras pessoas e disponibilizada para reutilização por meio de *pacotes*.

Go tem mais de cem pacotes padrões que oferecem a base para a maioria das aplicações. A comunidade Go – um ecossistema próspero de design, compartilhamento, reutilização e melhoria de pacotes – publicou muito mais, e você pode encontrar um índice em <http://godoc.org>, onde é possível fazer buscas. Neste capítulo, mostraremos como usar pacotes existentes e criar novos pacotes.

Go também vem acompanhado da ferramenta **go**: um comando sofisticado, porém fácil de usar, para administrar workspaces (espaços de trabalho) de pacotes Go. Desde o início do livro, mostramos como usar a ferramenta **go** para fazer download, build (construir) e executar programas de exemplo. Neste capítulo, daremos uma olhada nos conceitos subjacentes da ferramenta e conheceremos melhor suas funcionalidades, que incluem exibir documentação e consultar metadados dos pacotes no workspace. No próximo capítulo, exploraremos seus recursos de teste.

10.1 Introdução

O propósito de qualquer sistema de pacotes é tornar prático o design e a manutenção de programas grandes, agrupando funcionalidades relacionadas em unidades que possam ser compreendidas e alteradas, independentemente de outros pacotes do programa. Essa *modularidade*

permite que pacotes sejam compartilhados e reutilizados por projetos diferentes, distribuídos em uma empresa ou disponibilizados para o mundo.

Cada pacote define um espaço de nomes (name space) distinto que engloba seus identificadores. Cada nome é associado a um pacote em particular, permitindo a escolha de nomes concisos e claros para tipos, funções e outros itens que usamos com mais frequência, sem criar conflitos com outras partes do programa.

Os pacotes também oferecem *encapsulamento*, controlando quais nomes são visíveis ou exportados para fora do pacote. Restringir a visibilidade dos membros do pacote oculta as funções auxiliares e os tipos por trás da API do pacote, permitindo que o mantenedor mude a implementação com a certeza de que nenhum código fora do pacote será afetado. Restringir a visibilidade também oculta variáveis, de modo que os clientes possam acessá-las e atualizá-las somente por meio de funções exportadas que preservam as invariantes internas ou garantem exclusão mútua em um programa concorrente.

Quando alteramos um arquivo, devemos recompilar seu pacote e, potencialmente, todos os pacotes que dependam dele. A compilação de Go é notadamente mais rápida que a maioria das linguagens compiladas, mesmo quando fazemos o build do zero. Há três motivos principais para a velocidade do compilador. Em primeiro lugar, todas as importações devem ser explicitamente listadas no início de cada arquivo-fonte; portanto, o compilador não precisa ler e processar todo um arquivo para determinar suas dependências. Em segundo, as dependências de um pacote formam um grafo direcionado acíclico, e pelo fato de não haver ciclos, os pacotes podem ser compilados separadamente, talvez, em paralelo. Por fim, o arquivo-objeto de um pacote Go compilado registra informações de exportação não só do próprio pacote mas também de suas dependências. Ao compilar um pacote, o compilador precisa ler um arquivo-objeto para cada importação, mas não precisa olhar para além desses arquivos.

10.2 Caminhos de importação

Cada pacote é identificado por uma string única chamada *caminho de importação* (import path). Caminhos de importação são as strings que aparecem em declarações `import`.

```
import (  
    "fmt"  
    "math/rand"  
    "encoding/json"  
    "golang.org/x/net/html"  
    "github.com/go-sql-driver/mysql"  
)
```

Conforme mencionamos na seção 2.6.1, a especificação da linguagem Go não define o significado dessas strings nem como determinar o caminho de importação de um pacote, mas deixa essas questões a cargo das ferramentas. Neste capítulo, veremos detalhadamente como a ferramenta **go** as interpreta, pois é o que a maioria dos programadores Go usa para fazer build, testar, e assim por diante. No entanto, há outras ferramentas. Por exemplo, programadores Go que usam o sistema de build interno para múltiplas linguagens do Google seguem regras diferentes para nomear e localizar pacotes, especificar testes, entre outros, que se assemelham mais às convenções daquele sistema.

Para pacotes que você pretende compartilhar ou publicar, os caminhos de importação devem ser globalmente únicos. A fim de evitar conflitos, os caminhos de importação de todos os pacotes que não sejam os pacotes da biblioteca-padrão devem começar com o nome de domínio da organização na internet, que é o proprietário ou quem hospeda o pacote; isso permite que seja possível encontrar os pacotes. Por exemplo, a declaração anterior importa um parser HTML mantido pela equipe de Go e um driver popular de banco de dados MySQL de terceiros.

10.3 A declaração do pacote

Uma declaração **package** é necessária no início de todo arquivo-fonte Go.

Seu propósito principal é determinar o identificador default desse pacote (chamado de *nome do pacote*) quando ele é importado por outro pacote.

Por exemplo, todo arquivo do pacote **math/rand** começa com **package rand**; portanto, quando importar esse pacote, você poderá acessar seus membros como **rand.Int**, **rand.Float64**, e assim por diante.

```
package main

import (
    "fmt"
    "math/rand"
)

func main() {
    fmt.Println(rand.Int())
}
```

Por convenção, o nome do pacote é o último segmento do caminho de importação e, como resultado, dois pacotes podem ter o mesmo nome, apesar de seus caminhos de importação serem necessariamente diferentes. Por exemplo, os pacotes cujos caminhos de importação são **math/rand** e **crypto/rand** têm ambos o mesmo nome: **rand**. Veremos como usar os dois no mesmo programa em breve.

Há três exceções principais na convenção do “último segmento”. A primeira é que um pacote que defina um comando (um programa Go executável) sempre tem o nome **main**, independentemente do caminho de importação do pacote. Isso é um sinal para **go build** (seção 10.7.3) de que ele deve chamar o linker para criar um arquivo executável.

A segunda exceção é que alguns arquivos do diretório podem ter o sufixo **_test** em seus nomes de pacote se o nome do arquivo terminar com **_test.go**. Um diretório desse tipo pode definir *dois* pacotes: o pacote usual e outro chamado *pacote de testes externo*. O sufixo **_test** informa **go test** que ele deve construir ambos os pacotes e indica quais arquivos pertencem a cada um. Pacotes de testes externos são usados para evitar ciclos no grafo de importação resultante das dependências do teste: eles serão discutidos com mais detalhes na seção 11.2.4.

A terceira exceção é que algumas ferramentas para gerenciamento de

dependências concatenam sufixos com números de versão nos caminhos de importação de pacotes, como em `"gopkg.in/yaml.v2"`. O nome do pacote não inclui o sufixo; portanto, nesse caso, ele seria apenas `yaml`.

10.4 Declarações de importação

Um arquivo-fonte em Go pode conter zero ou mais declarações `import` imediatamente após a declaração `package` e antes da primeira declaração que não seja de importação. Cada declaração de importação pode especificar o caminho de importação de um único pacote ou de vários em uma lista entre parênteses. As duas formas a seguir são equivalentes, mas o segundo formato é mais comum.

```
import "fmt"
import "os"

import (
    "fmt"
    "os"
)
```

Pacotes importados podem ser agrupados introduzindo linhas em branco; esses agrupamentos geralmente indicam domínios diferentes. A ordem não é significativa, mas, por convenção, as linhas de cada grupo são especificadas em ordem alfabética. (Tanto `gofmt` quanto `goimports` farão o agrupamento e a ordenação para você.)

```
import (
    "fmt"
    "html/template"
    "os"

    "golang.org/x/net/html"
    "golang.org/x/net/ipv4"
)
```

Se precisarmos importar dois pacotes cujos nomes são iguais, como `math/rand` e `crypto/rand`, em um terceiro pacote, a declaração de importação deve especificar um nome alternativo para pelo menos um deles a fim de evitar conflito. Isso se chama *importar renomeando* (renaming import).

```
import (  
    "crypto/rand"  
    mrand "math/rand" // nome alternativo mrand evita conflito  
)
```

O nome alternativo afeta apenas o arquivo que faz a importação. Outros arquivos, até mesmo aqueles no mesmo pacote, podem importar o pacote usando seu nome default ou um nome diferente.

Importar renomeando pode ser útil mesmo quando não há conflitos. Se o nome do pacote importado é de difícil manejo, como ocorre às vezes no caso de código gerado de forma automática, um nome abreviado pode ser mais conveniente. O mesmo nome conciso deve ser usado consistentemente para evitar confusão. Escolher um nome alternativo pode ajudar a evitar conflitos com nomes comuns de variáveis locais. Por exemplo, em um arquivo com muitas variáveis locais de nome **path**, podemos importar o pacote padrão "**path**" como **pathpkg**.

Cada declaração de importação define uma dependência do pacote atual para o pacote importado. A ferramenta **go build** informa um erro se essas dependências formarem um ciclo.

10.5 Importações vazias

É um erro importar um pacote em um arquivo, mas não referenciar o nome que ele define nesse arquivo. No entanto, ocasionalmente precisamos importar um pacote apenas pelos efeitos colaterais dessa operação: a avaliação das expressões de inicialização de suas variáveis de nível de pacote e a execução de suas funções **init** (seção 2.6.2). Para coibir o erro de “importação não usada” que teríamos, devemos fazer uma importação que renomeie o pacote, em que o nome alternativo seja **_**, isto é, o identificador vazio. Como sempre, o identificador vazio jamais pode ser referenciado.

```
import _ "image/png" // registra o decodificador de PNG
```

Isso é conhecido como *importação vazia* (blank import). Ela é usada com mais frequência para implementar um mecanismo de tempo de

compilação em que o programa principal pode habilitar recursos opcionais ao fazer importações vazias de pacotes adicionais. Inicialmente veremos como usá-la e, em seguida, como ela funciona.

O pacote `image` da biblioteca-padrão exporta uma função `Decode` que lê bytes de um `io.Reader`, descobre qual formato de imagem foi usado para codificar os dados, chama o decodificador apropriado e então devolve a `image.Image` resultante. Usando `image.Decode` é fácil criar um conversor simples de imagem que leia uma imagem em um formato e a escreva em outro:

`gopl.io/ch10/jpeg`

```
// O comando jpeg lê uma imagem PNG da entrada-padrão
// e a escreve como uma imagem JPEG na saída-padrão.
package main

import (
    "fmt"
    "image"
    "image/jpeg"
    _ "image/png" // registra o decodificador de PNG
    "io"
    "os"
)

func main() {
    if err := toJPEG(os.Stdin, os.Stdout); err != nil {
        fmt.Fprintf(os.Stderr, "jpeg: %v\n", err)
        os.Exit(1)
    }
}

func toJPEG(in io.Reader, out io.Writer) error {
    img, kind, err := image.Decode(in)
    if err != nil {
        return err
    }
    fmt.Fprintln(os.Stderr, "Input format =", kind)
    return jpeg.Encode(out, img, &jpeg.Options{Quality: 95})
}
```

Se passarmos a saída de `gopl.io/ch3/mandelbrot` (seção 3.3) ao programa conversor, ele identificará o formato de entrada PNG e

escreverá uma versão JPEG da figura 3.3.

```
$ go build gopl.io/ch3/mandelbrot
$ go build gopl.io/ch10/jpeg
$ ./mandelbrot | ./jpeg >mandelbrot.jpg
Input format = png
```

Observe a importação vazia de `image/png`. Sem essa linha, o programa compila e faz link como sempre, mas não é mais capaz de reconhecer nem de decodificar entradas no formato PNG:

```
$ go build gopl.io/ch10/jpeg
$ ./mandelbrot | ./jpeg >mandelbrot.jpg
jpeg: image: unknown format
```

Eis o modo como isso funciona. A biblioteca-padrão oferece decodificadores para GIF, PNG e JPEG, e os usuários podem fornecer outros, mas para manter os executáveis pequenos, os decodificadores não são incluídos em uma aplicação, a menos que sejam explicitamente solicitados. A função `image.Decode` consulta uma tabela de formatos aceitos. Cada entrada da tabela especifica quatro informações: o nome do formato; uma string que é um prefixo para todas as imagens codificadas dessa maneira, usada para identificar a codificação; uma função `Decode` que decodifica uma imagem codificada; outra função `DecodeConfig` que decodifica apenas os metadados da imagem, como seu tamanho e o espaço de cores. Uma entrada é adicionada à tabela chamando `image.RegisterFormat`, geralmente a partir do inicializador do pacote que dá suporte a cada formato, como este em `image/png`:

```
package png // image/png

func Decode(r io.Reader) (image.Image, error)
func DecodeConfig(r io.Reader) (image.Config, error)

func init() {
    const pngHeader = "\x89PNG\r\n\x1a\n"
    image.RegisterFormat("png", pngHeader, Decode, DecodeConfig)
}
```

O efeito é que uma aplicação só precisa realizar uma importação vazia do pacote para o formato de que precisa para fazer a função `image.Decode` ser capaz de decodificá-la.

O pacote `database/sql` usa um mecanismo semelhante para permitir que os usuários instalem apenas os drivers de banco de dados necessários. Por exemplo:

```
import (
    "database/sql"
    _ "github.com/lib/pq"           // habilita suporte para Postgres
    _ "github.com/go-sql-driver/mysql" // habilita suporte para MySQL
)

db, err = sql.Open("postgres", dbname) // OK
db, err = sql.Open("mysql", dbname)    // OK
db, err = sql.Open("sqlite3", dbname)  // devolve erro: driver "sqlite3"
                                         // desconhecido
```

Exercício 10.1: Estenda o programa `jpeg` para que ele converta qualquer formato de entrada aceito para qualquer formato de saída, usando `image.Decode` para identificar o formato de entrada e uma flag para selecionar o formato de saída.

Exercício 10.2: Defina uma função genérica de leitura de arquivo capaz de ler arquivos ZIP (`archive/zip`) e arquivos tar POSIX (`archive/tar`). Use um sistema de registros semelhante àquele descrito anteriormente para que o suporte a cada formato de arquivo possa ser habilitado usando importações vazias.

10.6 Pacotes e nomenclatura

Nesta seção, ofereceremos alguns conselhos sobre como seguir convenções características de Go para nomear pacotes e seus membros.

Ao criar um pacote, mantenha seu nome conciso, mas não tão curto a ponto de ser enigmático. Os pacotes da biblioteca-padrão usados com mais frequência chamam-se `bufio`, `bytes`, `flag`, `fmt`, `http`, `io`, `json`, `os`, `sort`, `sync` e `time`.

Sempre que possível, seja descritivo e não ambíguo. Por exemplo, não chame um pacote utilitário de `util` quando um nome como `imageutil` ou `ioutil` é específico e ainda conciso. Evite escolher nomes de pacote que sejam comumente usados para variáveis locais relacionadas, ou você

forçará os clientes dele a importarem renomeando, como ocorre com o pacote **path**.

Nomes de pacote em geral assumem a forma singular. Os pacotes padrões **bytes**, **errors** e **strings** usam o plural para evitar que os tipos pré-declarados correspondentes sejam ocultos e, no caso de **go/types**, para evitar conflito com uma palavra reservada.

Evite nomes de pacote que já tenham outras conotações. Por exemplo, originalmente usamos o nome **temp** para o pacote de conversão de temperatura na seção 2.5, mas isso não durou muito. Foi uma péssima ideia, pois “temp” é quase um sinônimo universal para “temporário”. Passamos por um curto período com o nome **temperature**, mas o nome era longo demais e não dizia o que o pacote fazia. No final, tornou-se **tempconv**, que é menor e faz um paralelo com **strconv**.

Vamos agora nos voltar para a nomenclatura dos membros do pacote. Como cada referência a um membro de outro pacote usa um identificador qualificado como **fmt.Println**, a responsabilidade de descrever o membro do pacote pesa igualmente sobre o nome deste e o nome do membro. Não precisamos mencionar o conceito de formatação em **Println** porque o nome do pacote, **fmt**, já faz isso. Ao projetar um pacote, considere como as duas partes de um identificador qualificado funcionam em conjunto, e não o nome do membro sozinho. Eis alguns exemplos característicos:

```
bytes.Equal      flag.Int      http.Get      json.Marshal
```

Podemos identificar alguns padrões comuns de nomenclatura. O pacote **strings** oferece algumas funções independentes para manipulação de strings:

```
package strings

func Index(needle, haystack string) int

type Replacer struct{ /* ... */ }
func NewReplacer(oldnew ...string) *Replacer

type Reader struct{ /* ... */ }
func.NewReader(s string) *Reader
```

A palavra **string** não aparece em nenhum de seus nomes. Clientes referem-se a eles como **strings.Index**, **strings.Replacer**, e assim por diante.

Outros pacotes que podemos descrever como *pacotes de tipo único*, como **html/template** e **math/rand**, expõem um tipo de dado principal além de seus métodos, e, frequentemente, uma função **New** para criar instâncias.

```
package rand // "math/rand"

type Rand struct{ /* ... */ }
func New(source Source) *Rand
```

Isso pode resultar em repetição, como em **template.Template** ou em **rand.Rand**, motivo pelo qual os nomes desses tipos de pacotes muitas vezes são especialmente concisos.

Na outra extremidade, há pacotes como **net/http** que têm muitos nomes sem muitas estruturas porque realizam uma tarefa complexa. Apesar de ter mais de vinte tipos e muito mais funções, os membros mais importantes do pacote têm os nomes mais simples possíveis: **Get**, **Post**, **Handle**, **Error**, **Client**, **Server**.

10.7 Ferramenta go

O restante deste capítulo diz respeito à ferramenta **go**, usada para fazer download, consultar, formatar, fazer build, testar e instalar pacotes de código Go.

A ferramenta **go** combina os recursos de um conjunto variado de ferramentas em um só comando. É um gerenciador de pacotes (análogo a **apt** ou a **rpm**) que responde a consultas sobre seu conjunto de pacotes, processa suas dependências e faz download deles a partir de sistemas remotos de controle de versão. É um sistema de build que processa dependências de arquivos e chama compiladores, assemblers e linkers, embora seja propositalmente menos completo que o **make** padrão do Unix. Além disso, é um executor de testes, como veremos no capítulo 11.

Sua interface de linha de comando usa o estilo de “canivete suíço”, com

mais de uma dúzia de subcomandos, alguns dos quais já vimos, como **get**, **run**, **build** e **fmt**. Você pode executar **go help** para ver o índice de sua documentação embutida, mas, como referência, listamos os comandos mais comuns a seguir:

```
$ go
...
build      compile packages and dependencies
clean      remove object files
doc        show documentation for package or symbol
env        print Go environment information
fmt        run gofmt on package sources
get        download and install packages and dependencies
install    compile and install packages and dependencies
list       list packages
run        compile and run Go program
test       test packages
version    print Go version
vet        run go tool vet on packages

Use "go help [command]" for more information about a command.
...
```

Para manter a necessidade de configuração em um nível mínimo, a ferramenta **go** se apoia fortemente em convenções. Por exemplo, dado o nome de um arquivo-fonte em Go, a ferramenta pode encontrar o pacote que o inclui, pois cada diretório contém um único pacote, e o caminho de importação de um pacote corresponde à hierarquia de diretórios no workspace. Dado o caminho de importação de um pacote, a ferramenta é capaz de encontrar o diretório correspondente em que os arquivos-objeto são armazenados. Ela também é capaz de encontrar o URL do servidor que hospeda o repositório de código-fonte.

10.7.1 Organização do workspace

A única configuração que a maioria dos usuários precisa é da variável de ambiente **GOPATH**, que especifica a raiz do workspace¹. Ao mudar para um workspace diferente, os usuários atualizam o valor de **GOPATH**. Por exemplo, definimos **GOPATH** com **\$HOME/gobook** enquanto trabalhamos

neste livro:

```
$ export GOPATH=$HOME/gobook
$ go get gopl.io/...
```

Depois que fizer o download de todos os programas deste livro usando o comando anterior, seu workspace conterá uma hierarquia como esta:

```
GOPATH/
  src/
    gopl.io/
      .git/
      ch1/
        helloworld/
          main.go
        dup/
          main.go
        ...
    golang.org/x/net/
      .git/
      html/
        parse.go
        node.go
        ...
  bin/
    helloworld
    dup
  pkg/
    darwin_amd64/
    ...
```

GOPATH tem três subdiretórios. O subdiretório **src** armazena o código-fonte. Cada pacote está em um diretório cujo nome relativo a **\$GOPATH/src** é o caminho de importação do pacote, por exemplo, **gopl.io/ch1/helloworld**. Observe que um único workspace **GOPATH** contém vários repositórios de controle de versão abaixo de **src**, como **gopl.io** ou **golang.org**. O subdiretório **pkg** é o local em que as ferramentas de build armazenam pacotes compilados, e o subdiretório **bin** armazena programas executáveis, como **helloworld**.

Uma segunda variável de ambiente, **GOROOT**, especifica o diretório-raiz da distribuição de Go, que disponibiliza todos os pacotes da biblioteca-

padrão. A estrutura de diretórios abaixo de **GOROOT** lembra a estrutura de **GOPATH**. Assim, por exemplo, os arquivos-fontes do pacote **fmt** estão no diretório **\$GOROOT/src/fmt**. Os usuários não precisam definir **GOROOT**, pois, por padrão, a ferramenta **go** usará a localização em que ela foi instalada.

O comando **go env** exibe os valores das variáveis de ambiente relevantes à cadeia de ferramentas, incluindo os valores default das variáveis ausentes. **GOOS** especifica o sistema operacional alvo (por exemplo, **android**, **linux**, **darwin** ou **windows**) e **GOARCH** especifica a arquitetura do processador-alvo, como **amd64**, **386** ou **arm**. Embora **GOPATH** seja a única variável que você deve definir², as outras variáveis ocasionalmente aparecem em nossas explicações.

```
$ go env
GOPATH="/home/gopher/gobook"
GOROOT="/usr/local/go"
GOARCH="amd64"
GOOS="darwin"
...
```

10.7.2 Fazendo download de pacotes

Quando usar a ferramenta **go**, o caminho de importação de um pacote informa não só o lugar em que ele se encontra no workspace local mas também onde encontrá-lo na internet, de modo que **go get** possa obtê-lo e atualizá-lo.

O comando **go get** pode fazer download de um único pacote ou de uma subárvore completa ou repositório usando a notação **...**, como na seção anterior. A ferramenta também processa e faz download de todas as dependências dos pacotes iniciais, motivo pelo qual o pacote **golang.org/x/net/html** apareceu no workspace no exemplo anterior.

Depois que **go get** tiver feito download dos pacotes, ele faz seu build e *instala* as bibliotecas e os comandos. Veremos os detalhes na próxima seção, mas um exemplo mostrará como o processo é simples. O primeiro comando a seguir obtém a ferramenta **golint**, que verifica problemas

comuns de estilo em código-fonte Go. O segundo comando executa **golint** em **gopl.io/ch2/popcount** da seção 2.6.2. Ela informa prestativamente que esquecemos de escrever um comentário doc (doc comment) para o pacote:

```
$ go get github.com/golang/lint/golint
$ $GOPATH/bin/golint gopl.io/ch2/popcount
src/gopl.io/ch2/popcount/main.go:1:1:
    package comment should be of the form "Package popcount ..."
```

O comando **go get** tem suporte para sites populares de hospedagem como GitHub, Bitbucket e Launchpad, e pode fazer as requisições apropriadas aos seus sistemas de controle de versões. Para sites menos conhecidos, talvez você precise informar o protocolo de controle de versões a ser usado no caminho de importação, por exemplo, Git ou Mercurial. Execute **go help importpath** para ver os detalhes.

Os diretórios que **go get** cria são verdadeiros clones do repositório remoto, e não apenas cópias dos arquivos; portanto, você pode usar comandos de controle de versão para ver um diff de edições locais feitas ou atualizar para uma versão diferente. Por exemplo, o diretório **golang.org/x/net** é um clone de um repositório Git:

```
$ cd $GOPATH/src/golang.org/x/net
$ git remote -v
origin https://go.googlesource.com/net (fetch)
origin https://go.googlesource.com/net (push)
```

Observe que o nome de domínio aparente no caminho de importação do pacote, **golang.org**, difere do verdadeiro nome de domínio do servidor Git, **go.googlesource.com**. Esse é um recurso da ferramenta **go** que permite que pacotes usem um nome de domínio personalizado em seu caminho de importação, ao mesmo tempo em que são hospedados por um serviço genérico como **googlesource.com** ou **github.com**. Páginas HTML abaixo de **https://golang.org/x/net/html** incluem os metadados mostrados a seguir, que redirecionam a ferramenta **go** para o repositório Git no site de hospedagem propriamente dito:

```
$ go build gopl.io/ch1/fetch
$ ./fetch https://golang.org/x/net/html | grep go-import
```



```
<meta name="goimport"
  content="golang.org/x/net git https://go.goglesource.com/net">
```

Se você especificar a flag **-u**, **go get** garantirá que todos os pacotes que visitar, incluindo as dependências, serão atualizados para sua versão mais recente antes de fazer o build e a instalação. Sem essa flag, os pacotes que já existem localmente não serão atualizados.

O comando **go get -u** em geral recupera a versão mais recente de cada pacote, o que é conveniente quando você está começando, mas pode não ser apropriado para projetos implantados em produção, em que um controle preciso das dependências é crucial para uma implantação saudável. A solução usual para esse problema é o *vendoring* (fornecimento) do código, isto é, fazer uma cópia local persistente de todas as dependências necessárias e atualizar essa cópia de forma cuidadosa e deliberada. Antes da versão 1.5 de Go, isso exigia mudar os caminhos de importação desses pacotes, portanto, nossa cópia de golang.org/x/net/html se transformava em gopl.io/vendor/golang.org/x/net/html. Versões mais recentes da ferramenta **go** tratam vendoring diretamente, embora não tenhamos espaço para mostrar os detalhes aqui. Veja *Vendor Directories* na saída do comando **go help gopath**.

Exercício 10.3: Usando **fetch http://gopl.io/ch1/helloworld?go-get=1**, descubra qual serviço hospeda os exemplos de código deste livro. (Requisições HTTP de **go get** incluem o parâmetro **go-get** para que os servidores possam distingui-los de requisições comuns de navegador.)

10.7.3 Build de pacotes

O comando **go build** compila cada pacote especificado como argumento. Se o pacote for uma biblioteca, o resultado é descartado; isso simplesmente verifica se o pacote está livre de erros de compilação. Se o pacote se chamar **main**, **go build** chama o linker para criar um executável no diretório atual; o nome do executável é obtido a partir do último segmento do caminho de importação do pacote.

Como cada diretório contém um pacote, cada programa executável – ou *comando* na terminologia Unix – exige seu próprio diretório. Esses diretórios às vezes são filhos de um diretório chamado **cmd**, como no comando golang.org/x/tools/cmd/godoc, que serve a documentação de pacotes Go por meio de uma interface web (seção 10.7.4).

Pacotes podem ser especificados por seus caminhos de importação, como vimos antes, ou por um nome de diretório relativo, que deve começar com um segmento `.` ou `..`, mesmo que isso não seja comumente necessário. Se nenhum argumento for fornecido, o diretório atual é usado. Assim, os comandos a seguir constroem o mesmo pacote, embora cada um escreva o executável no diretório em que **go build** é executado:

```
$ cd $GOPATH/src/gopl.io/ch1/helloworld
$ go build
```

e:

```
$ cd algum_lugar
$ go build gopl.io/ch1/helloworld
```

e:

```
$ cd $GOPATH
$ go build ./src/gopl.io/ch1/helloworld
```

mas não:

```
$ cd $GOPATH
$ go build src/gopl.io/ch1/helloworld
Error: cannot find package "src/gopl.io/ch1/helloworld".
```

Pacotes também podem ser especificados como uma lista de nomes de arquivos, embora a tendência seja usar isso somente para programas pequenos e experimentos realizados apenas uma vez. Se o nome do pacote for **main**, o nome do executável será extraído do nome-base do primeiro arquivo **.go**.

```
$ cat quoteargs.go
package main

import (
    "fmt"
    "os"
)
```

```
func main() {  
    fmt.Printf("%q\n", os.Args[1:])  
}  
$ go build quoteargs.go  
$ ./quoteargs one "two three" four\ five  
["one" "two three" "four five"]
```

Particularmente, para programas descartáveis como esse, queremos rodar o executável assim que ele for construído. O comando **go run** combina os dois passos a seguir:

```
$ go run quoteargs.go one "two three" four\ five  
["one" "two three" "four five"]
```

O comando **go run** assume que o primeiro argumento que não termine com **.go** seja o início da lista de argumentos para o executável Go.

Por padrão, o comando **go build** constrói o pacote solicitado e todas as suas dependências e, então, descarta todo o código compilado, exceto o executável final, se houver. Tanto a análise de dependência quanto a compilação são surpreendentemente rápidos, porém, à medida que os projetos crescem para dezenas de pacotes e centenas de milhares de linhas de código, o tempo para recompilar as dependências pode se tornar perceptível, podendo durar vários segundos, mesmo quando essas dependências não sofreram nenhuma alteração.

O comando **go install** é bem semelhante a **go build**, exceto por salvar o código compilado de cada pacote e comando em vez de descartá-lo. Pacotes compilados são salvos abaixo do diretório **\$GOPATH/pkg** correspondente ao diretório **src** em que o código-fonte está, e comandos executáveis são salvos no diretório **\$GOPATH/bin**. (Muitos usuários colocam **\$GOPATH/bin** no caminho de busca de executáveis do shell.) Desse modo, **go build** e **go install** não executam o compilador para esses pacotes e comandos se eles não sofreram alterações, tornando os builds subsequentes muito mais rápidos. Por conveniência, **go build -i** instala os pacotes que são dependências do alvo do build.

Como pacotes compilados variam de acordo com a plataforma e a arquitetura, **go install** os salva abaixo de um subdiretório cujo nome incorpora os valores das variáveis de ambiente **GOOS** e **GOARCH**. Por

exemplo, em um Mac, o pacote `golang.org/x/net/html` é compilado e instalado no arquivo `golang.org/x/net/html.a` em `$GOPATH/pkg/darwin_amd64`.

É simples fazer uma *cross-compilação* de um programa Go, isto é, construir um executável destinado a um sistema operacional ou a uma CPU diferente. Basta definir as variáveis `GOOS` ou `GOARCH` durante o build. O programa `cross` exibe o sistema operacional e a arquitetura para os quais o build foi feito:

gopl.io/ch10/cross

```
func main() {  
    fmt.Println(runtime.GOOS, runtime.GOARCH)  
}
```

Os comandos a seguir geram executáveis para 64 bits e 32 bits, respectivamente:

```
$ go build gopl.io/ch10/cross  
$ ./cross  
darwin amd64  
$ GOARCH=386 go build gopl.io/ch10/cross  
$ ./cross  
darwin 386
```

Alguns pacotes podem precisar compilar versões diferentes de código para determinadas plataformas ou processadores a fim de lidar com questões de portabilidade de baixo nível ou para oferecer versões otimizadas de rotinas importantes, por exemplo. Se o nome de um arquivo inclui um nome de sistema operacional ou de arquitetura de processador, como `net_linux.go` ou `asm_amd64.s`, a ferramenta `go` compilará o arquivo somente quando fizer build para esse alvo. Comentários especiais chamados *tags de build* (build tags) permitem um controle mais minucioso. Por exemplo, se um arquivo contém este comentário:

```
// +build linux darwin
```

antes da declaração do pacote (e de seu comentário doc), `go build` o compilará somente quando fizer build para Linux ou para Mac OS X, e o comentário a seguir diz para não compilar o arquivo:

```
// +build ignore
```

Para mais detalhes, veja a seção *Build Constraints* (Restrições de build) na documentação do pacote **go/build**:

```
$ go doc go/build
```

10.7.4 Documentando pacotes

O bom estilo em Go incentiva fortemente uma boa documentação das APIs dos pacotes. Cada declaração de um membro exportado do pacote e a própria declaração deste devem ser imediatamente precedidas por um comentário que explica seus propósito e uso.

Os *comentários doc* (doc comments) de Go são sempre frases completas, e a primeira delas geralmente é um resumo que começa com o nome sendo declarado. Os parâmetros de função e outros identificadores são mencionados sem aspas ou marcação. Por exemplo, eis o comentário doc de **fmt.Fprintf**:

```
// Fprintf formats according to a format specifier and writes to w.  
// It returns the number of bytes written and any write error encountered.  
func Fprintf(w io.Writer, format string, a ...interface{}) (int, error)
```

Os detalhes da formatação de **Fprintf** estão explicados em um comentário doc associado ao próprio pacote **fmt**. Um comentário que anteceda imediatamente uma declaração **package** é considerado o comentário doc do pacote como um todo. Deve haver apenas um, embora ele possa aparecer em qualquer arquivo. Comentários mais longos de pacote podem merecer um arquivo próprio; o comentário de **fmt** tem mais de 300 linhas. Esse arquivo em geral se chama **doc.go**.

Uma boa documentação não precisa ser extensa, e documentação não substitui simplicidade. De fato, é convencional em Go buscar a concisão e a simplicidade na documentação, como em tudo, pois a documentação, assim como o código, também exige manutenção. Muitas declarações podem ser explicadas em uma frase bem redigida e, se o comportamento for realmente óbvio, nenhum comentário será necessário.

Ao longo deste livro, conforme o espaço permitiu, inserimos comentários

doc antes de várias declarações, mas você encontrará melhores exemplos à medida que navegar pela biblioteca-padrão. Duas ferramentas podem ajudar a fazer isso.

A ferramenta **go doc** exibe a declaração e o comentário doc da entidade especificada na linha de comando, que pode ser um pacote:

```
$ go doc time
package time // importa "time"

Package time provides functionality for measuring and displaying time.

const Nanosecond Duration = 1 ...
func After(d Duration) <-chan Time
func Sleep(d Duration)
func Since(t Time) Duration
func Now() Time
type Duration int64
type Time struct { ... }
...vários outros...
```

ou um membro de pacote:

```
$ go doc time.Since
func Since(t Time) Duration

    Since returns the time elapsed since t.
    It is shorthand for time.Now().Sub(t).
```

ou um método:

```
$ go doc time.Duration.Seconds
func (d Duration) Seconds() float64

    Seconds returns the duration as a floating-point number of seconds.
```

A ferramenta não precisa de caminhos de importação completos nem especificação exata de letras maiúsculas ou minúsculas. O comando a seguir exibe a documentação de **(*json.Decoder).Decode** do pacote **encoding/json**:

```
$ go doc json.decoder
func (dec *Decoder) Decode(v interface{}) error

    Decode reads the next JSON-encoded value from its input and stores
    it in the value pointed to by v.
```

A segunda ferramenta, cujo nome **godoc** pode causar confusão, serve

páginas HTML associadas que oferecem as mesmas informações que **godoc** e muito mais. O servidor **godoc** em <https://golang.org/pkg> inclui a biblioteca-padrão. A figura 10.1 mostra a documentação do pacote **time** e, na seção 11.6, veremos a exibição interativa de programas de exemplo feita por **godoc**. O servidor **godoc** em <https://godoc.org> tem um índice de milhares de pacotes de código aberto em que é possível fazer buscas.

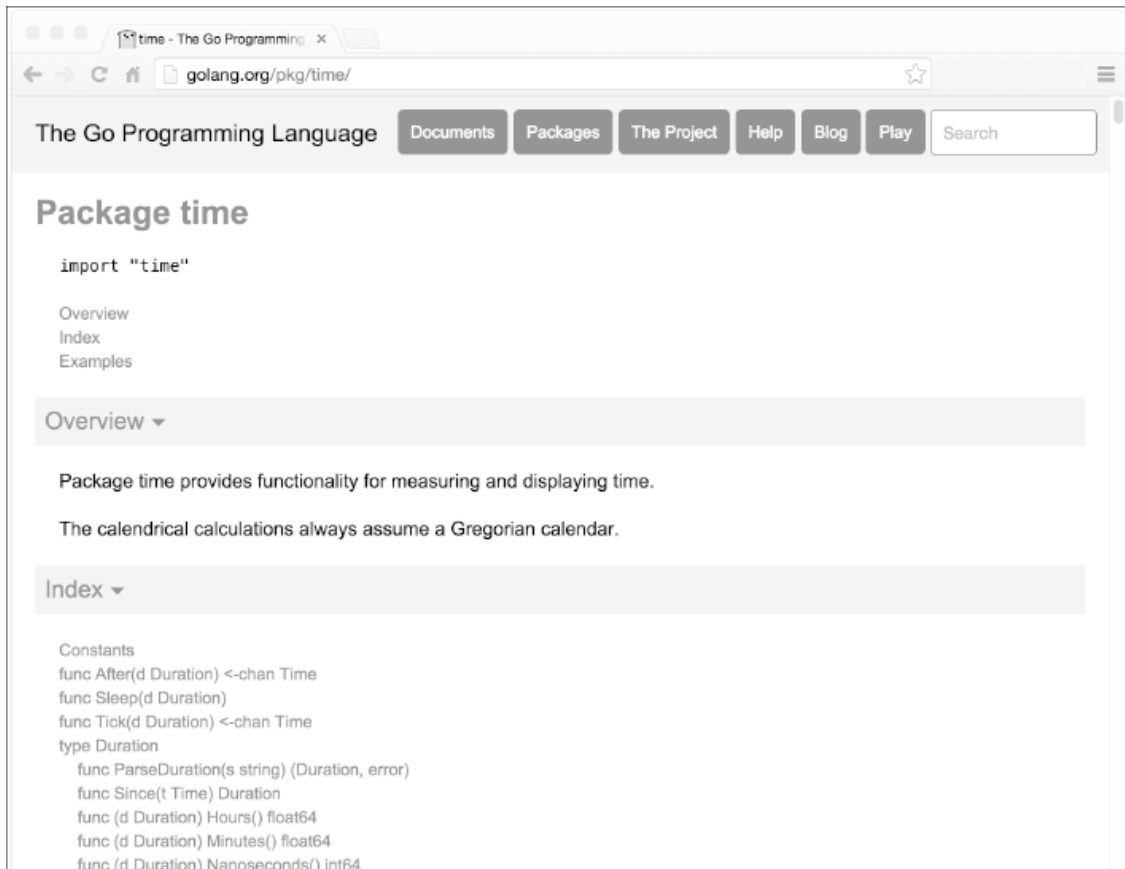


Figura 10.1 – O pacote time em godoc.

Você também pode executar uma instância de **godoc** em seu workspace se quiser navegar em seus próprios pacotes. Acesse <http://localhost:8000/pkg> em seu navegador enquanto executa este comando:

```
$ godoc -http :8000
```

Suas flags **-analysis=type** e **-analysis=pointer** expandem a documentação e o código-fonte com os resultados de análise estática avançada.

10.7.5 Pacotes internos

O pacote é o sistema mais importante de encapsulamento em programas Go. Identificadores não exportados são visíveis somente dentro do mesmo pacote, enquanto identificadores exportados são visíveis ao mundo.

Às vezes, porém, uma solução intermediária seria útil: uma maneira de definir identificadores que sejam visíveis a um pequeno conjunto de pacotes confiáveis, mas não a todos. Por exemplo, quando estivermos dividindo um pacote grande em partes mais administráveis, talvez não desejemos revelar as interfaces entre essas partes a outros pacotes. Ou podemos querer compartilhar funções utilitárias entre vários pacotes de um projeto sem expô-los de forma mais ampla. Ou, quem sabe, só queremos fazer experimentos com um novo pacote sem nos comprometermos prematuramente com sua API, colocando-o em “período de experiência” com um conjunto limitado de clientes.

Para atender a essas necessidades, a ferramenta **go build** trata um pacote de modo especial se seu caminho de importação contiver um segmento de path chamado **internal**. Esses pacotes são chamados de *pacotes internos* (internal packages). Um pacote interno pode ser importado somente por outro que esteja dentro da árvore cuja raiz seja o pai do diretório **internal**. Por exemplo, dados os pacotes a seguir, **net/http/internal/chunked** pode ser importado de **net/http/httputil** ou de **net/http**, mas não de **net/url**. Contudo, **net/url** pode importar **net/http/httputil**.

```
net/http
net/http/internal/chunked
net/http/httputil
net/url
```

10.7.6 Consulta de pacotes

A ferramenta **go list** dá informações sobre pacotes disponíveis. Em sua forma mais simples, **go list** testa se um pacote está presente no workspace e exibe seu caminho de importação em caso afirmativo:


```
$ go list github.com/go-sql-driver/mysql
github.com/go-sql-driver/mysql
```

Um argumento para **go list** pode conter o curinga "...", que casa com qualquer substring do caminho de importação de um pacote. Podemos usá-lo para listar todos os pacotes em um workspace de Go:

```
$ go list ...
archive/tar
archive/zip
bufio
bytes
cmd/addr2line
cmd/api
...vários outros...
```

ou em uma subárvore específica:

```
$ go list gopl.io/ch3/...
gopl.io/ch3/basename1
gopl.io/ch3/basename2
gopl.io/ch3/comma
gopl.io/ch3/mandelbrot
gopl.io/ch3/netflag
gopl.io/ch3/printints
gopl.io/ch3/surface
```

ou listar os pacotes relacionados a um assunto em particular:

```
$ go list ...xml...
encoding/xml
gopl.io/ch7/xmlselect
```

O comando **go list** obtém todos os metadados de cada pacote, e não apenas o caminho de importação, e disponibiliza essas informações aos usuários ou a outras ferramentas em diversos formatos. A flag **-json** faz **go list** exibir o registro completo de cada pacote em formato JSON:

```
$ go list -json hash
{
  "Dir": "/home/gopher/go/src/hash",
  "ImportPath": "hash",
  "Name": "hash",
  "Doc": "Package hash provides interfaces for hash functions.",
  "Target": "/home/gopher/go/pkg/darwin_amd64/hash.a",
  "Goroot": true,
```

```

    "Standard": true,
    "Root": "/home/gopher/go",
    "GoFiles": [
        "hash.go"
    ],
    "Imports": [
        "io"
    ],
    "Deps": [
        "errors",
        "io",
        "runtime",
        "sync",
        "sync/atomic",
        "unsafe"
    ]
}

```

A flag **-f** permite que os usuários personalizem o formato da saída usando a linguagem de template do pacote **text/template** (seção 4.6). O comando a seguir exibe as dependências transitivas do pacote **strconv**, separadas por espaços:

```

$ go list -f '{{join .Deps " "}}' strconv
errors math runtime unicode/utf8 unsafe

```

e o próximo comando exibe as importações diretas de cada pacote na subárvore **compress** da biblioteca-padrão:

```

$ go list -f '{{.ImportPath}} -> {{join .Imports " "}}' compress/...
compress/bzip2 -> bufio io sort
compress/flate -> bufio fmt io math sort strconv
compress/gzip -> bufio compress/flate errors fmt hash hash/crc32 io time
compress/lzw -> bufio errors fmt io
compress/zlib -> bufio compress/flate errors fmt hash hash/adler32 io

```

O comando **go list** é útil tanto para consultas interativas feitas apenas uma vez quanto para construir e testar scripts de automação. Vamos usá-lo novamente na seção 11.2.4. Para obter mais informações, incluindo o conjunto de campos disponíveis e seus significados, veja a saída de **go help list**.

Neste capítulo, explicamos todos os subcomandos importantes da

ferramenta **go**, exceto um. No próximo capítulo, veremos como o comando **go test** é usado para testar programas Go.

Exercício 10.4: Construa uma ferramenta que informe o conjunto de todos os pacotes do workspace que dependam transitivamente dos pacotes especificados pelos argumentos. Dica: você precisará executar **go list** duas vezes, uma para os pacotes iniciais e outra para todos os pacotes. Talvez você queira fazer parse de sua saída JSON usando o pacote **encoding/json** (seção 4.5).

1 Nota do Revisor da Tradução: a partir de Go 1.8 não é mais preciso definir **GOPATH**. Na ausência desta variável de ambiente, a ferramenta go assume o valor default que é **\$HOME/go** em sistemas Unix e **%USERPROFILE%\go** no Windows.

2 N.R.T.: Opcional a partir de Go 1.8. Veja mais informações na nota anterior.

11

Testes

Maurice Wilkes, desenvolvedor do EDSAC, o primeiro computador com programa armazenado, teve um insight surpreendente enquanto subia a escada de seu laboratório em 1949. No livro *Memoirs of a Computer Pioneer*, ele recorda o seguinte: “Tive a forte percepção de que passaria boa parte do restante de minha vida encontrando erros em meus próprios programas.”¹ Certamente, todo programador de computador com programa armazenado desde então pode se solidarizar com Wilkes, embora, talvez, não sem certa dose de perplexidade pela sua ingenuidade sobre as dificuldades da construção de um software.

Programas atualmente são bem maiores e mais complexos que os da época de Wilkes, é claro, e uma boa dose de esforços tem sido gasta em técnicas para deixar essa complexidade administrável. Duas técnicas em particular se destacam pela sua eficácia. A primeira é a rotineira revisão por pares (peer review) de programas antes que eles sejam implantados. A segunda, que é o assunto deste capítulo, são testes.

Testes, pelo que implicitamente queremos dizer testes *automatizados*, são a prática de escrever pequenos programas que verificam se o código em teste (o código de *produção*) comporta-se conforme esperado para determinadas entradas que, em geral, são cuidadosamente escolhidas a fim de exercitar certas funcionalidades ou são aleatórias para garantir uma ampla cobertura.

O campo dos testes de software é enorme. A tarefa de testar ocupa parte do tempo de todos os programadores e todo o tempo de alguns programadores. A literatura sobre testes inclui milhares de livros impressos e milhões de palavras em postagens de blog. Em toda linguagem de programação importante há dezenas de pacotes de software

para construção de testes, alguns com bastante teoria, e o campo parece atrair uns tantos profetas com seguidores quase devotos. É quase suficiente para convencer programadores de que, para escrever testes eficientes, é obrigatório adquirir um conjunto totalmente novo de habilidades.

A abordagem de Go para testes pode parecer um tanto low tech quando comparada a outras. Ela depende de um só comando, **go test**, e de um conjunto de convenções para escrever funções de teste que **go test** possa executar. O mecanismo comparativamente leve é eficaz para testes puros e se estende de modo natural para benchmarks e exemplos sistemáticos para documentação.

Na prática, escrever código de teste não é muito diferente de escrever o próprio programa original. Escrevemos funções pequenas que focam em uma parte da tarefa. Devemos ter cuidado com condições limítrofes, pensar nas estruturas de dados e pensar nos resultados que um processamento deve gerar a partir de entradas adequadas. Porém, esse é o mesmo processo usado para escrever código comum em Go; não é necessário usar novas notações, convenções ou ferramentas.

11.1 A ferramenta go test

O subcomando **go test** é um executor de testes para pacotes Go organizados de acordo com determinadas convenções. Em um diretório de pacote, arquivos cujos nomes terminem com **_test.go** não fazem parte do pacote comumente construído por **go build**, mas fazem parte dele quando construídos por **go test**.

Em arquivos ***_test.go**, três tipos de função são tratados de modo especial: testes, benchmarks e exemplos. Uma *função de teste* (test function), que é uma função cujo nome começa com **Test**, exercita uma lógica do programa para verificar se seu comportamento está correto; **go test** chama a função de teste e informa o resultado, que é **PASS** ou **FAIL**. Uma *função de benchmark* (benchmark function) tem um nome que começa com **Benchmark**, e mede o desempenho de alguma operação; **go**

test informa o tempo médio de execução da operação. Uma *função de exemplo* (example function), cujo nome começa com **Example**, suporta documentação verificada pela máquina. Abordaremos os testes em detalhes na seção 11.2, os benchmarks na seção 11.4 e os exemplos na seção 11.6.

A ferramenta **go test** percorre os arquivos ***_test.go** em busca dessas funções especiais, gera um pacote **main** temporário que as chama de maneira apropriada, faz o build e o executa, informa os resultados e, então, faz a limpeza.

11.2 Funções Test

Todo arquivo de teste deve importar o pacote **testing**. As funções de teste têm a seguinte assinatura:

```
func TestNome(t *testing.T) {  
    // ...  
}
```

Os nomes das funções de teste devem começar com **Test**; o sufixo opcional *Nome* deve começar com uma letra maiúscula:

```
func TestSin(t *testing.T) { /* ... */ }  
func TestCos(t *testing.T) { /* ... */ }  
func TestLog(t *testing.T) { /* ... */ }
```

O parâmetro **t** oferece métodos para informar falhas de teste e fazer logging de informações adicionais. Vamos definir um pacote **gopl.io/ch11/word1** de exemplo contendo uma única função **IsPalindrome** que informa se uma string é lida do mesmo jeito de frente para trás e de trás para frente. (Esta implementação testa todos os bytes duas vezes se a string for um palíndromo; voltaremos a esse assunto em breve.)

gopl.io/ch11/word1

```
// O pacote word oferece utilitários para jogos de palavra.  
package word  
  
// IsPalindrome informa se s é lida do mesmo jeito de frente para trás e  
// de trás para a frente.
```

```
// (Nossa primeira tentativa.)
func IsPalindrome(s string) bool {
    for i := range s {
        if s[i] != s[len(s)-1-i] {
            return false
        }
    }
    return true
}
```

No mesmo diretório, o arquivo `word_test.go` contém duas funções de teste chamadas `TestPalindrome` e `TestNonPalindrome`. Cada uma verifica se `IsPalindrome` dá a resposta correta para uma única entrada e informa falhas usando `t.Error`:

```
package word

import "testing"

func TestPalindrome(t *testing.T) {
    if !IsPalindrome("detartrated") {
        t.Error(`IsPalindrome("detartrated") = false`)
    }
    if !IsPalindrome("kayak") {
        t.Error(`IsPalindrome("kayak") = false`)
    }
}

func TestNonPalindrome(t *testing.T) {
    if IsPalindrome("palindrome") {
        t.Error(`IsPalindrome("palindrome") = true`)
    }
}
```

Um comando `go test` (ou `go build`) sem argumentos de pacote atua no pacote do diretório atual. Podemos fazer build e executar os testes com o seguinte comando:

```
$ cd $GOPATH/src/gopl.io/ch11/word1
$ go test
ok   gopl.io/ch11/word1 0.008s
```

Satisfeitos, lançamos o programa, mas assim que os convidados da festa de lançamento saem, os relatórios de bug começam a chegar. Uma usuária francesa chamada Noelle Eve Elleon reclama que `IsPalindrome` não

reconhece “été”. Outra usuária da América Central está desapontada porque o programa rejeita “A man, a plan, a canal: Panama”. Esses relatórios de bug específicos e pequenos naturalmente resultam em novos casos de teste.

```
func TestFrenchPalindrome(t *testing.T) {
    if !IsPalindrome("été") {
        t.Error(`IsPalindrome("été") = false`)
    }
}

func TestCanalPalindrome(t *testing.T) {
    input := "A man, a plan, a canal: Panama"
    if !IsPalindrome(input) {
        t.Errorf(`IsPalindrome(%q) = false`, input)
    }
}
```

Para evitar escrever a longa string `input` duas vezes, usamos `Errorf`, que oferece formatação como `Printf`.

Quando os dois novos testes forem adicionados, o comando `go test` falha com mensagens de erro informativas.

```
$ go test
--- FAIL: TestFrenchPalindrome (0.00s)
    word_test.go:28: IsPalindrome("été") = false
--- FAIL: TestCanalPalindrome (0.00s)
    word_test.go:35: IsPalindrome("A man, a plan, a canal: Panama") = false
FAIL
FAIL    gopl.io/ch11/word1 0.014s
```

É uma boa prática escrever o teste antes e observar se ele dispara a mesma falha descrita no relatório de bug do usuário. Somente então podemos estar confiantes de que, qualquer que seja a correção que elaborarmos, ela abordará o problema certo.

Como bônus, executar `go test` em geral é mais rápido do que executar manualmente os passos descritos no relatório de bug, permitindo fazer uma iteração mais rápida. Se a suíte de testes contiver muitos passos lentos, podemos fazer um progresso mais rápido ainda se formos seletivos em relação àqueles que vamos executar.

A flag `-v` exibe o nome e o tempo de execução de cada teste do pacote:

```
$ go test -v
=== RUN TestPalindrome
--- PASS: TestPalindrome (0.00s)
=== RUN TestNonPalindrome
--- PASS: TestNonPalindrome (0.00s)
=== RUN TestFrenchPalindrome
--- FAIL: TestFrenchPalindrome (0.00s)
    word_test.go:28: IsPalindrome("été") = false
=== RUN TestCanalPalindrome
--- FAIL: TestCanalPalindrome (0.00s)
    word_test.go:35: IsPalindrome("A man, a plan, a canal: Panama") = false
FAIL
exit status 1
FAIL    gopl.io/ch11/word1 0.017s
```

e a flag `-run`, cujo argumento é uma expressão regular, faz `go test` executar somente os testes cujo nome de função case com o padrão:

```
$ go test -v -run="French|Canal"
=== RUN TestFrenchPalindrome
--- FAIL: TestFrenchPalindrome (0.00s)
    word_test.go:28: IsPalindrome("été") = false
=== RUN TestCanalPalindrome
--- FAIL: TestCanalPalindrome (0.00s)
    word_test.go:35: IsPalindrome("A man, a plan, a canal: Panama") = false
FAIL
exit status 1
FAIL    gopl.io/ch11/word1 0.014s
```

É claro que, depois que os testes selecionados passarem, devemos chamar `go test` sem flags para executar a suíte de testes completa uma última vez antes de fazer commit das alterações.

Agora nossa tarefa é corrigir os bugs. Uma investigação rápida revela que a causa do primeiro bug é o uso de sequências de bytes por `IsPalindrome`, e não de sequências de runas, de modo que caracteres não ASCII como `é` em `"été"` o confundem. O segundo bug surge porque a função não ignora espaços, pontuação e a diferença entre letras maiúsculas e minúsculas.

Humildemente, reescrevemos a função com mais cuidado:

gopl.io/ch11/word2

```
// O pacote word oferece utilitários para jogos de palavra.
package word

import "unicode"

// IsPalindrome informa se s é lida da mesma maneira de frente para trás e
// de trás para a frente.
// A diferença entre letras maiúsculas e minúsculas é ignorada, assim como
// os caracteres que não são letras.
func IsPalindrome(s string) bool {
    var letters []rune
    for _, r := range s {
        if unicode.IsLetter(r) {
            letters = append(letters, unicode.ToLower(r))
        }
    }
    for i := range letters {
        if letters[i] != letters[len(letters)-1-i] {
            return false
        }
    }
    return true
}
```

Também escrevemos um conjunto mais abrangente de casos de teste que combina todos os anteriores com alguns casos novos em uma tabela.

```
func TestIsPalindrome(t *testing.T) {
    var tests = []struct {
        input string
        want bool
    }{
        {"", true},
        {"a", true},
        {"aa", true},
        {"ab", false},
        {"kayak", true},
        {"detartrated", true},
        {"A man, a plan, a canal: Panama", true},
        {"Evil I did dwell; lewd did I live.", true},
        {"Able was I ere I saw Elba", true},
        {"été", true},
        {"Et se resservir, ivresse reste.", true},
        {"palindrome", false}, // não é palíndromo
    }
```

```

    {"desserts", false}, // semi-palíndromo2
}
for _, test := range tests {
    if got := IsPalindrome(test.input); got != test.want {
        t.Errorf("IsPalindrome(%q) = %v", test.input, got)
    }
}
}

```

Nossos novos testes passam:

```

$ go test gopl.io/ch11/word2
ok      gopl.io/ch11/word2      0.015s

```

Esse estilo de testes *orientados a tabela* (table-driven) é muito comum em Go. É fácil acrescentar novas entradas na tabela conforme for necessário e, como a lógica de asserção não é duplicada, podemos investir mais esforços em gerar uma boa mensagem de erro.

A saída de um teste com falha *não* inclui toda a stack trace no momento da chamada a `t.Errorf`. Tampouco `t.Errorf` gera um pânico ou interrompe a execução do teste, diferente das falhas de asserção em muitos frameworks de teste para outras linguagens. Os testes são independentes uns dos outros. Se uma entrada anterior da tabela fizer o teste falhar, entradas seguintes na tabela continuarão a ser verificadas e, desse modo, poderemos obter informações sobre várias falhas durante uma única execução.

Quando tivermos realmente que interromper uma função de teste, talvez porque algum código de inicialização falhou ou para impedir que uma falha já informada cause uma cascata confusa de outras, usamos `t.Fatal` ou `t.Fatalf`. Elas devem ser chamados da mesma gorrotina que a função `Test`, e não de outra criada durante o teste.

As mensagens de falhas de teste geralmente estão no formato "`f(x) = y, want z`" ("`f(x) = y, esperado z`"), em que `f(x)` explica a operação que se tentou executar e sua entrada, `y` é o resultado obtido e `z` é o resultado esperado. Quando for conveniente, como em nosso exemplo de palíndromo, a sintaxe de Go é usada para a parte `f(x)`. Exibir `x` é especialmente importante em um teste orientado a tabela, pois uma dada

asserção é executada muitas vezes com valores diferentes. Evite código repetitivo (boilerplate) e informações redundantes. Quando testar uma função booleana como `IsPalindrome`, omita a parte `want z`, pois ela não acrescenta nenhuma informação. Se `x`, `y` ou `z` forem extensos, exiba um resumo das partes relevantes em seu lugar. O autor de um teste deve se esforçar para ajudar o programador que precisa diagnosticar uma falha em um teste.

Exercício 11.1: Escreva testes para o programa `charcount` da seção 4.3.

Exercício 11.2: Escreva um conjunto de testes para `IntSet` (seção 6.5) que verifique se seu comportamento após cada operação é equivalente a um conjunto baseado em mapas embutidos. Guarde sua implementação para benchmarking no exercício 11.7.

11.2.1 Testes aleatórios

Testes orientados a tabela são convenientes para verificar se uma função está correta para entradas cuidadosamente selecionadas e para exercitar casos interessantes de lógica. Outra abordagem, os *testes aleatórios* (randomized testing), explora uma variedade maior de entradas construindo entradas aleatoriamente.

Como sabemos qual é a saída esperada de nossa função, dada uma entrada aleatória? Existem duas estratégias. A primeira é escrever uma implementação alternativa da função que use um algoritmo menos eficiente, porém mais simples e mais claro, e verificar se as duas implementações produzem o mesmo resultado. A segunda é criar valores de entrada de acordo com um padrão para que possamos saber qual é a saída esperada.

O exemplo a seguir usa a segunda abordagem: a função `randomPalindrome` gera palavras construídas para serem palíndromos.

```
import "math/rand"

// randomPalindrome devolve um palíndromo cujo tamanho e conteúdo
// são derivados do gerador de números pseudoaleatórios rng.
func randomPalindrome(rng *rand.Rand) string {
```

```

    n := rng.Intn(25) // tamanho aleatório até 24
    runes := make([]rune, n)
    for i := 0; i < (n+1)/2; i++ {
        r := rune(rng.Intn(0x1000)) // runa aleatória até '\u0999'
        runes[i] = r
        runes[n-1-i] = r
    }
    return string(runes)
}

func TestRandomPalindromes(t *testing.T) {
    // Inicializa um gerador de números pseudoaleatórios.
    seed := time.Now().UTC().UnixNano()
    t.Logf("Random seed: %d", seed)
    rng := rand.New(rand.NewSource(seed))
    for i := 0; i < 1000; i++ {
        p := randomPalindrome(rng)
        if !IsPalindrome(p) {
            t.Errorf("IsPalindrome(%q) = false", p)
        }
    }
}

```

Como testes aleatórios são não determinísticos, é fundamental que o log do teste com falha registre informações suficientes para reproduzir a falha. Em nosso exemplo, a entrada `p` de `IsPalindrome` informa tudo que precisamos saber, mas para funções que aceitem entradas mais complexas, pode ser mais fácil fazer log da semente (seed) do gerador de números pseudoaleatórios (como vimos antes) do que fazer dump de toda a estrutura de dados de entrada. De posse desse valor de semente, podemos facilmente modificar o teste a fim de reproduzir a falha de modo determinístico.

Ao usar o horário atual como fonte de aleatoriedade, o teste explorará novas entradas a cada vez que executar, durante todo seu tempo de vida. Isso é importante em especial se seu projeto usa um sistema automatizado para executar todos os testes periodicamente.

Exercício 11.3: `TestRandomPalindromes` testa somente palíndromos. Escreva um teste aleatório que gere e verifique *não* palíndromos.

Exercício 11.4: Modifique `randomPalindrome` para exercitar o tratamento de

pontuações e espaços por `IsPalindrome`.

11.2.2 Testando um comando

A ferramenta `go test` é útil para testar pacotes de biblioteca, mas, com um pouco de esforço, podemos usá-la para testar comandos também. Um pacote chamado `main` frequentemente gera um programa executável, mas também pode ser importado como uma biblioteca.

Vamos escrever um teste para o programa `echo` da seção 2.3.2. Dividimos o programa em duas funções: `echo` faz o verdadeiro trabalho, enquanto `main` faz parse e lê os valores de flag, além de informar qualquer erro devolvido por `echo`.

gopl.io/ch11/echo

```
// Echo exhibe seus argumentos de linha de comando.
package main

import (
    "flag"
    "fmt"
    "io"
    "os"
    "strings"
)

var (
    n = flag.Bool("n", false, "omit trailing newline")
    s = flag.String("s", " ", "separator")
)

var out io.Writer = os.Stdout // modificado durante os testes

func main() {
    flag.Parse()
    if err := echo(!*n, *s, flag.Args()); err != nil {
        fmt.Fprintf(os.Stderr, "echo: %v\n", err)
        os.Exit(1)
    }
}

func echo(newline bool, sep string, args []string) error {
    fmt.Fprint(out, strings.Join(args, sep))
    if newline {
```

```

        fmt.Fprintln(out)
    }
    return nil
}

```

A partir do teste, chamaremos **echo** com vários argumentos e configurações de flag e verificaremos se ele exibe a saída correta em cada caso; portanto, adicionamos parâmetros a **echo** a fim de reduzir sua dependência de variáveis globais. Apesar do que dissemos, também introduzimos outra variável global **out**, que é o **io.Writer** no qual o resultado será escrito. Ao fazer **echo** escrever por meio dessa variável, e não diretamente em **os.Stdout**, os testes podem fazer uma substituição por uma implementação diferente de **Writer** que registre o que foi escrito para uma inspeção posterior. Eis o teste no arquivo **echo_test.go**:

```

package main

import (
    "bytes"
    "fmt"
    "testing"
)

func TestEcho(t *testing.T) {
    var tests = []struct {
        newline bool
        sep      string
        args     []string
        want     string
    }{
        {true, "", []string{}, "\n"},
        {false, "", []string{}, ""},
        {true, "\t", []string{"one", "two", "three"}, "one\ttwo\tthree\n"},
        {true, ",", []string{"a", "b", "c"}, "a,b,c\n"},
        {false, ":", []string{"1", "2", "3"}, "1:2:3"},
    }
    for _, test := range tests {
        descr := fmt.Sprintf("echo(%v, %q, %q)",
            test.newline, test.sep, test.args)
        out = new(bytes.Buffer) // saída capturada
        if err := echo(test.newline, test.sep, test.args); err != nil {
            t.Errorf("%s failed: %v", descr, err)
            continue
        }
    }
}

```

```

    }
    got := out.(*bytes.Buffer).String()
    if got != test.want {
        t.Errorf("%s = %q, want %q", descr, got, test.want)
    }
}
}

```

Observe que o código de teste está no mesmo pacote que o código de produção. Embora o nome do pacote seja **main** e defina uma função **main**, durante os testes, esse pacote atua como uma biblioteca que expõe a função **TestEcho** ao executor de testes; sua função **main** é ignorada.

Ao organizar o teste na forma de uma tabela, podemos facilmente acrescentar novos casos de teste. Vamos ver o que acontece quando o teste falha, adicionando a linha a seguir à tabela:

```

{true, ",", []string{"a", "b", "c"}, "a b c\n"}, // NOTA: expectativa
                                                    // incorreta!

```

go test exibe:

```

$ go test gopl.io/ch11/echo
--- FAIL: TestEcho (0.00s)
    echo_test.go:31: echo(true, ",", ["a" "b" "c"]) = "a,b,c", want "a b c\n"
FAIL
FAIL    gopl.io/ch11/echo    0.006s

```

A mensagem de erro descreve a operação que se tentou executar (usando uma sintaxe semelhante a Go), o comportamento propriamente dito e o comportamento esperado, nessa ordem. Com uma mensagem de erro informativa como essa, você pode ter uma ideia muito boa da causa-raiz antes mesmo de localizar o código-fonte do teste.

É importante que o código que está sendo testado não chame **log.Fatal** nem **os.Exit**, pois elas interromperão o processo de imediato; chamar essas funções deve ser um direito exclusivo de **main**. Se algo totalmente inesperado acontecer e uma função gerar pânico, o executor de testes se recuperará, embora o teste, é claro, seja considerado uma falha. Erros esperados como aqueles resultantes de uma entrada ruim do usuário, arquivos ausentes ou configuração indevida devem ser informados pela devolução de um valor **error** diferente de **nil**. Felizmente (embora

infelizmente como ilustração), nosso exemplo com **echo** é tão simples que jamais retornará um erro diferente de nil.

11.2.3 Testes caixa-branca

Uma maneira de classificar testes é de acordo com o nível de conhecimento que eles exigem do funcionamento interno do pacote em teste. Um teste *caixa-preta* (black-box) não supõe nada sobre o pacote além do que é exposto pela sua API e especificado pela sua documentação; a parte interna do pacote é opaca. Em comparação, um teste *caixa-branca* (white-box) tem acesso privilegiado às funções internas e às estruturas de dados do pacote e pode fazer observações e alterações que um cliente comum não pode. Por exemplo, um teste caixa-branca pode verificar se as invariantes dos tipos de dados do pacote são preservadas após cada operação. [O nome *caixa-branca* é tradicional, mas *caixa-clara* (clear box) seria mais preciso.]

As duas abordagens são complementares. Testes caixa-preta geralmente são mais robustos e precisam de menos atualizações à medida que o software evolui. Eles também ajudam o autor dos testes a ter empatia com o cliente do pacote e podem revelar falhas no design da API. Por outro lado, testes caixa-branca podem oferecer uma cobertura mais detalhada das partes mais intrincadas da implementação.

Já vimos exemplos dos dois tipos. **TestIsPalindrome** chama somente a função exportada **IsPalindrome** e, assim, é um teste caixa-preta. **TestEcho** chama a função **echo** e atualiza a variável global **out**; ambas não são exportadas, o que faz do teste um teste caixa-branca.

Enquanto desenvolvíamos **TestEcho**, modificamos a função **echo** para que usasse a variável de nível de pacote **out** ao escrever sua saída. Assim o teste pode substituir a saída-padrão por uma implementação alternativa que registre os dados para uma inspeção posterior. Usando a mesma técnica, podemos substituir outras partes do código de produção com implementações “forjadas” fáceis de testar. A vantagem das implementações forjadas é que elas podem ser mais simples de configurar,

mais previsíveis, mais confiáveis e mais fáceis de observar. Elas podem também evitar efeitos colaterais indesejáveis, como atualizar um banco de dados de produção ou fazer cobrança em um cartão de crédito.

O código a seguir mostra a lógica de verificação de quotas em um web service que oferece armazenamento em rede para usuários. Quando os usuários excedem 90% de sua quota, o sistema lhes envia um e-mail de aviso.

gopl.io/ch11/storage1

```
package storage

import (
    "fmt"
    "log"
    "net/smtp"
)

var usage = make(map[string]int64)

func bytesInUse(username string) int64 { return usage[username] }

// Configuração para envio de e-mail.
// NOTA: jamais coloque senhas no códigos-fonte!
const sender = "notifications@example.com"
const password = "correcthorsebatterystaple"
const hostname = "smtp.example.com"

const template = `Warning: you are using %d bytes of storage,
%d%% of your quota.`

func CheckQuota(username string) {
    used := bytesInUse(username)
    const quota = 1000000000 // 1GB
    percent := 100 * used / quota
    if percent < 90 {
        return // OK
    }
    msg := fmt.Sprintf(template, used, percent)
    auth := smtp.PlainAuth("", sender, password, hostname)
    err := smtp.SendMail(hostname+":587", auth, sender,
        []string{username}, []byte(msg))
    if err != nil {
        log.Printf("smtp.SendMail(%s) failed: %s", username, err)
    }
}
```

```
}
```

Gostaríamos de testá-lo, mas não queremos que o teste envie um e-mail de verdade. Portanto, transferimos a lógica de e-mail para sua própria função e a armazenamos em uma variável não exportada no nível de pacote, `notifyUser`.

gopl.io/ch11/storage2

```
var notifyUser = func(username, msg string) {
    auth := smtp.PlainAuth("", sender, password, hostname)
    err := smtp.SendMail(hostname+":587", auth, sender,
        []string{username}, []byte(msg))
    if err != nil {
        log.Printf("smtp.SendEmail(%s) failed: %s", username, err)
    }
}

func CheckQuota(username string) {
    used := bytesInUse(username)
    const quota = 1000000000 // 1GB
    percent := 100 * used / quota
    if percent < 90 {
        return // OK
    }
    msg := fmt.Sprintf(template, used, percent)
    notifyUser(username, msg)
}
```

Agora podemos escrever um teste que faz a substituição do envio de e-mail de verdade por um sistema simulado simples de notificação. Esse sistema registra o usuário notificado e o conteúdo da mensagem.

```
package storage

import (
    "strings"
    "testing"
)

func TestCheckQuotaNotifiesUser(t *testing.T) {
    var notifiedUser, notifiedMsg string
    notifyUser = func(user, msg string) {
        notifiedUser, notifiedMsg = user, msg
    }
    const user = "joe@example.org"
```

```

usage[user] = 980000000 // simula uma condição de 980 MB usados
CheckQuota(user)
if notifiedUser == "" && notifiedMsg == "" {
    t.Fatalf("notifyUser not called")
}
if notifiedUser != user {
    t.Errorf("wrong user (%s) notified, want %s",
        notifiedUser, user)
}
const wantSubstring = "98% of your quota"
if !strings.Contains(notifiedMsg, wantSubstring) {
    t.Errorf("unexpected notification message <<%s>>, "+
        "want substring %q", notifiedMsg, wantSubstring)
}
}

```

Há um problema: depois que essa função de teste retorna, **CheckQuota** já não funciona como deveria porque continua usando a implementação simulada de **notifyUsers** para teste. (Há sempre um risco desse tipo quando atualizamos variáveis globais.) Devemos modificar o teste para que restaure o valor anterior, de modo que testes subsequentes não sofram nenhum efeito, e devemos fazer isso em todos os caminhos de execução, incluindo nas falhas e pânicos nos testes. Isso naturalmente sugere o uso de **defer**.

```

func TestCheckQuotaNotifiesUser(t *testing.T) {
    // Salva e restaura o notifyUser original.
    saved := notifyUser
    defer func() { notifyUser = saved }()

    // Instala o notifyUser simulado para teste.
    var notifiedUser, notifiedMsg string
    notifyUser = func(user, msg string) {
        notifiedUser, notifiedMsg = user, msg
    }
    // ...restante do teste...
}

```

Esse padrão pode ser usado para salvar temporariamente e restaurar todo tipo de variáveis globais, incluindo flags de linha de comando, opções de depuração e parâmetros de desempenho, para instalar e remover hooks que façam o código de produção chamar um código de teste quando algo

interessante acontece e para forçar o código de produção a estados raros, porém importantes, como timeouts, erros e até mesmo entrelaçamentos específicos de atividades concorrentes.

Usar variáveis globais dessa maneira só é seguro porque **go test** geralmente não executa vários testes de forma concorrente.

11.2.4 Pacotes de testes externos

Considere os pacotes **net/url**, que oferecem um parser de URL, e **net/http**, que oferecem um servidor web e uma biblioteca de cliente HTTP. Como poderíamos esperar, os pacotes **net/http** de mais alto nível dependem de **net/url** de mais baixo nível. No entanto, um dos testes em **net/url** é um exemplo que mostra a interação entre URLs e a biblioteca de cliente HTTP. Em outras palavras, um teste do pacote de mais baixo nível importa o pacote de mais alto nível.

Declarar essa função de teste no pacote **net/url** criaria um ciclo no grafo de importações de pacotes, conforme representado pela seta para cima na figura 11.1. Entretanto, como explicamos na seção 10.1, a especificação de Go proíbe ciclos de importação.

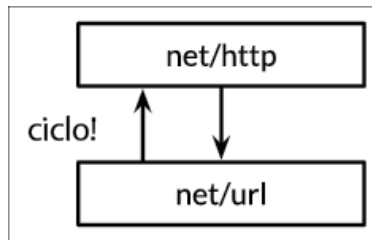


Figura 11.1 – Um teste de net/url depende de net/http.

Resolvemos o problema declarando a função de teste em um *pacote de testes externo*, isto é, em um arquivo no diretório **net/url** em cuja declaração de pacote lê-se **package url_test**. O sufixo extra **_test** é uma indicação para **go test** de que ele deve construir um pacote adicional contendo apenas esses arquivos e executar seus testes. Pensar nesse pacote de testes externo como se ele tivesse o path de importação **net/url_test** pode ajudar, mas ele não pode ser importado com esse nem com nenhum outro nome.

Como os testes externos estão em um pacote separado, eles podem importar pacotes auxiliares que também dependam do pacote que está sendo testado; um teste in-package (no pacote) não pode fazer isso. Em termos de camadas de design, o pacote de testes externo está logicamente acima de ambos os pacotes dos quais ele depende, como mostra a figura 11.2.

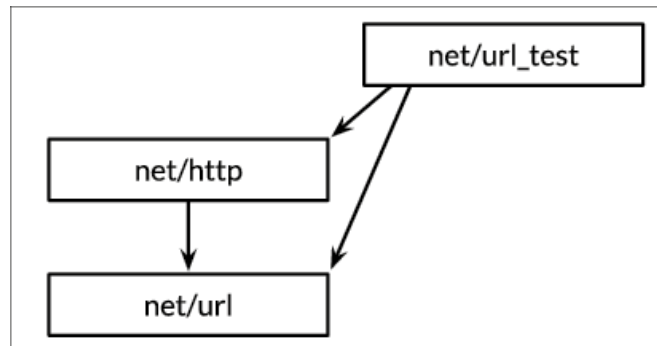


Figura 11.2 – Pacotes de testes externos quebram ciclos de dependência.

Ao evitar ciclos de importação, pacotes de testes externos permitem que testes, em especial *testes de integração* (que testam a interação entre vários componentes), importem outros pacotes livremente, como faria uma aplicação.

Podemos usar a ferramenta **go list** para resumir quais arquivos-fontes Go em um diretório de pacote são código de produção, testes in-package e testes externos. Usaremos o pacote **fmt** como exemplo. **GoFiles** é a lista de arquivos que contêm o código de produção; são os arquivos que **go build** incluirá em sua aplicação:

```
$ go list -f={{.GoFiles}} fmt
[doc.go format.go print.go scan.go]
```

TestGoFiles é a lista de arquivos que também pertencem ao pacote **fmt**, mas esses arquivos, cujos nomes terminam todos com **_test.go**, são incluídos somente no build dos testes:

```
$ go list -f={{.TestGoFiles}} fmt
[export_test.go]
```

Os testes do pacote geralmente ficariam nesses arquivos, embora, de modo incomum, **fmt** não tenha nenhum; explicaremos o propósito de

`export_test.go` em breve.

`XTestGoFiles` é a lista dos arquivos que compõem o pacote de testes externo, `fmt_test`. Portanto, esses arquivos devem importar o pacote `fmt` para usá-lo. Novamente, eles são incluídos somente durante os testes:

```
$ go list -f={{.XTestGoFiles}} fmt
[fmt_test.go scan_test.go stringer_test.go]
```

Às vezes, um pacote de testes externo pode precisar de acesso privilegiado à parte interna do pacote em teste se, por exemplo, um teste caixa-branca precisar estar em um pacote separado para evitar um ciclo de importação. Em casos como esse, usamos um truque: acrescentamos declarações em um arquivo in-package `_test.go` para expor a parte interna necessária ao teste externo. Assim, esse arquivo oferece ao teste uma “porta dos fundos” para o pacote. Se o arquivo-fonte existe apenas para esse propósito e não contém nenhum teste, com frequência ele é chamado de `export_test.go`.

Por exemplo, a implementação do pacote `fmt` precisa da funcionalidade de `unicode.IsSpace` como parte de `fmt.Scanf`. Para evitar criar uma dependência indesejável, `fmt` não importa o pacote `unicode` e suas grandes tabelas de dados; em vez disso, contém uma implementação mais simples, que chama de `isSpace`.

Para garantir que os comportamentos de `fmt.isSpace` e de `unicode.IsSpace` não se distanciem, `fmt` prudentemente contém um teste. É um teste externo e, assim, não pode acessar `isSpace` de forma direta, portanto, `fmt` abre uma porta dos fundos para ela declarando uma variável exportada que armazena a função `isSpace` interna. O código a seguir é o arquivo `export_test.go` completo do pacote `fmt`.

```
package fmt

var IsSpace = isSpace
```

Esse arquivo de teste não define nenhum teste; ele simplesmente declara o símbolo exportado `fmt.IsSpace` para ser usado pelo teste externo. Esse truque também é válido sempre que um teste externo precisar usar algumas das técnicas de testes caixa-branca.

11.2.5 Escrevendo testes eficazes

Muitos iniciantes em Go se surpreendem com o minimalismo do framework de testes de Go. Frameworks de outras linguagens oferecem mecanismos para identificar funções de teste (com frequência usando reflexão ou metadados), hooks para executar operações de “setup” (configuração) e “teardown” (restauração) antes e depois dos testes executarem e bibliotecas de funções utilitárias para asserção de predicados comuns, comparação de valores, formatação de mensagens de erro e para abortar um teste com falha (frequentemente usando exceções). Embora esses mecanismos possam deixar os testes bem concisos, os testes resultantes muitas vezes parecem ter sido escritos em uma linguagem diferente. Além do mais, embora os testes possam informar **PASS** ou **FAIL** corretamente, seu comportamento pode não ser amigável para o infeliz mantenedor, com mensagens de falha enigmáticas como `"assert: 0 == 1"` ou páginas e mais páginas de stack traces.

A atitude de Go diante dos testes mostra um forte contraste. Ela espera que autores de teste façam a maior parte desse trabalho por conta própria, definindo funções para evitar repetição, como fariam em programas comuns. O processo de testar não é como um preenchimento de formulário rotineiro; um teste também tem uma interface de usuário, embora seus únicos usuários são também seus mantenedores. Um bom teste não explode com falhas, mas exibe uma descrição clara e sucinta do sintoma do problema e, talvez, outros fatos relevantes sobre o contexto. O ideal é que o mantenedor não precise ler o código-fonte para decifrar uma falha de teste. Um bom teste não deve desistir após uma falha, mas tentar informar vários erros em uma única execução, pois o padrão de falhas, por si só, pode ser revelador.

A função de asserção a seguir compara dois valores, constrói uma mensagem de erro genérica e interrompe o programa. É fácil de usar e está correta, mas, quando falha, a mensagem de erro é quase inútil. Ela não resolve o difícil problema de oferecer uma boa interface de usuário.

```
import (  
    "fmt"
```



```

    "strings"
    "testing"
)
// Uma função de asserção precária.
func assertEqual(x, y int) {
    if x != y {
        panic(fmt.Sprintf("%d != %d", x, y))
    }
}
func TestSplit(t *testing.T) {
    words := strings.Split("a:b:c", ":")
    assertEqual(len(words), 3)
    // ...
}

```

Nesse sentido, funções de asserção sofrem de *abstração prematura*: ao tratar a falha desse teste em particular como uma simples diferença entre dois inteiros, perdemos a oportunidade de oferecer um contexto significativo. Podemos oferecer uma mensagem melhor começando pelos detalhes concretos, como no exemplo a seguir. Somente quando padrões repetitivos emergirem em uma dada suíte de testes será hora de introduzir abstrações.

```

func TestSplit(t *testing.T) {
    s, sep := "a:b:c", ":"
    words := strings.Split(s, sep)
    if got, want := len(words), 3; got != want {
        t.Errorf("Split(%q, %q) returned %d words, want %d",
            s, sep, got, want)
    }
    // ...
}

```

Agora o teste informa a função que foi chamada, suas entradas e a significância do resultado; ele identifica de modo explícito o valor propriamente dito e a expectativa e continua a executar, mesmo quando essa asserção falha. Após termos escrito um teste como esse, o próximo passo natural muitas vezes não é definir uma função para substituir toda a instrução `if`, mas executar o teste em um loop em que `s`, `sep` e `want` variem, como o teste orientado a tabela de `IsPalindrome`.

O exemplo anterior não precisou de nenhuma função utilitária, mas é claro que isso não nos impediria de introduzir funções quando elas ajudam a deixar o código mais simples. (Verificaremos uma função utilitária desse tipo, `reflect.DeepEqual`, na seção 13.3.) A chave para um bom teste é começar implementando o comportamento concreto que você quer e somente então usar funções para simplificar o código e eliminar repetições. Os melhores resultados raramente são obtidos começando com uma biblioteca de funções abstratas e genéricas de teste.

Exercício 11.5: Estenda `TestSplit` para que use uma tabela de entradas e de saídas esperadas.

11.2.6 Evitando testes frágeis

Uma aplicação que frequentemente falha quando encontra entradas novas, porém válidas, é chamada de *buggy* (infestada de bugs); um teste que falha de forma espúria quando uma alteração correta foi feita no programa chama-se *frágil* (brittle). Assim como um programa cheio de bugs frustra seus usuários, um teste frágil deixa seus mantenedores exasperados. Os testes mais frágeis, que falham para quase toda alteração no código de produção, boa ou ruim, às vezes são apelidados de *detectores de mudança* (change detectors) ou *testes de status quo* (*status quo* tests), e o tempo gasto para lidar com eles pode rapidamente acabar com qualquer vantagem que eles pareciam proporcionar.

Quando uma função em teste gera uma saída complexa, por exemplo, uma string longa, uma estrutura de dados elaborada ou um arquivo, é tentador verificar se a saída é exatamente igual a algum valor “de ouro” (golden) esperado quando o teste foi escrito. Porém, conforme o programa evolui, é provável que parte da saída mude para melhor, mas de alguma forma mude. E não é só a saída; funções com entradas complexas com frequência quebram porque a entrada usada em um teste não é mais válida.

A maneira mais simples de evitar testes frágeis é verificar somente as propriedades em que você está interessado. Prefira testar as interfaces mais

simples e mais estáveis de seu programa em vez de testar suas funções internas. Seja seletivo em suas asserções. Não verifique correspondências exatas de strings, por exemplo, mas procure substrings relevantes que permanecerão inalteradas à medida que o programa evoluir. Muitas vezes vale a pena escrever uma função não trivial para reduzir uma saída complexa à sua essência, de modo que as asserções sejam confiáveis. Mesmo que isso possa parecer bastante esforço prévio, pode compensar rapidamente em termos de tempo que, de outro modo, seria gasto corrigindo falhas espúrias em testes.

11.3 Cobertura

Pela sua natureza, os testes nunca terminam. Como o influente cientista de computação Edsger Dijkstra afirmou, “os testes mostram a presença, e não a ausência de bugs”. Nenhuma quantidade de testes é capaz de provar que um pacote está livre de bugs. No melhor caso, eles aumentam nossa confiança de que o pacote funciona bem em uma grande variedade de cenários importantes.

O grau com que uma suíte de testes exercita o pacote em teste chama-se *cobertura* (coverage) dos testes. A cobertura não pode ser diretamente quantificada – a dinâmica até mesmo dos programas mais triviais está além de uma medida exata –, mas há métodos heurísticos que podem ajudar a direcionar nossos esforços de teste para onde eles provavelmente sejam mais úteis.

A cobertura de instruções (statement coverage) é o método heurístico mais simples e mais amplamente usado. A cobertura de instruções de uma suíte de testes é a fração das instruções do código-fonte executada pelo menos uma vez durante o teste. Nesta seção, usaremos a ferramenta **cover** de Go, que está integrada a **go test**, para medir a cobertura de instruções e ajudar a identificar lacunas óbvias nos testes.

O código a seguir é um teste orientado a tabela para o avaliador de expressões que desenvolvemos no capítulo 7:

gopl.io/ch7/eval

```

func TestCoverage(t *testing.T) {
    var tests = []struct {
        input string
        env    Env
        want  string // erro esperado de Parse/Check ou resultado de Eval
    }{
        {"x % 2", nil, "unexpected '%'"},
        {"!true", nil, "unexpected '!'"},
        {"log(10)", nil, `unknown function "log"`},
        {"sqrt(1, 2)", nil, "call to sqrt has 2 args, want 1"},
        {"sqrt(A / pi)", Env{"A": 87616, "pi": math.Pi}, "167"},
        {"pow(x, 3) + pow(y, 3)", Env{"x": 9, "y": 10}, "1729"},
        {"5 / 9 * (F - 32)", Env{"F": -40}, "-40"},
    }
    for _, test := range tests {
        expr, err := Parse(test.input)
        if err == nil {
            err = expr.Check(map[Var]bool{})
        }
        if err != nil {
            if err.Error() != test.want {
                t.Errorf("%s: got %q, want %q", test.input, err, test.want)
            }
            continue
        }
        got := fmt.Sprintf("%.6g", expr.Eval(test.env))
        if got != test.want {
            t.Errorf("%s: %v => %s, want %s",
                test.input, test.env, got, test.want)
        }
    }
}

```

Inicialmente vamos verificar se o teste passa:

```

$ go test -v -run=Coverage gopl.io/ch7/eval
=== RUN TestCoverage
--- PASS: TestCoverage (0.00s)
PASS
ok      gopl.io/ch7/eval 0.011s

```

Este comando exibe a mensagem de uso da ferramenta de cobertura:

```

$ go tool cover
Usage of 'go tool cover':

```

Given a coverage profile produced by 'go test':

```
go test -coverprofile=c.out
```

Open a web browser displaying annotated source code:

```
go tool cover -html=c.out
```

...

O comando `go tool` roda um dos executáveis da cadeia de ferramentas de Go. Esses programas estão no diretório `$GOROOT/pkg/tool/${GOOS}_${GOARCH}`. Graças a `go build` é raro precisarmos chamá-las diretamente.

Agora executamos o teste com a flag `-coverprofile`:

```
$ go test -run=Coverage -coverprofile=c.out gopl.io/ch7/eval
ok      gopl.io/ch7/eval    0.032s coverage: 68.5% of statements
```

Essa flag habilita a coleta de dados de cobertura *instrumentando* o código de produção, isto é, ela modifica uma cópia do código-fonte para que, antes de cada bloco de instruções ser executado, uma variável booleana seja definida, com uma variável por bloco. Imediatamente antes de o programa modificado sair, ele escreve o valor de cada variável no arquivo de log especificado `c.out` e exibe um resumo da fração de instruções executada. (Se tudo de que você precisa é do resumo, use `go test -cover`.)

Se `go test` for executado com a flag `-covermode=count`, a instrumentação de cada bloco incrementa um contador em vez de definir um booleano. O log resultante com a contagem de execução de cada bloco permite fazer comparações quantitativa entre blocos “mais quentes”, isto é, mais frequentemente executados, e blocos “mais frios”.

Após ter coletado os dados, executamos a ferramenta `cover`, que processa o log, gera um relatório HTML e abre-o em uma nova janela do navegador (Figura 11.3).

```
$ go tool cover -html=c.out
```

Cada instrução é apresentada em verde se foi alcançada, ou em vermelho se não foi. Por questões de clareza, sombreamos o plano de fundo do texto em vermelho. Podemos ver imediatamente que nenhuma de nossas entradas exercitou o método `Eval` dos operadores unários. Se

acrescentarmos o novo caso de teste a seguir à tabela e executarmos os dois comandos anteriores novamente, o código para expressões unárias torna-se verde:

```
{"+x * -x", Env{"x": 2}, "4"}
```

Contudo as duas instruções **panic** permanecem em vermelho. Isso não deve ser nenhuma surpresa, pois espera-se que essas instruções sejam inacessíveis.

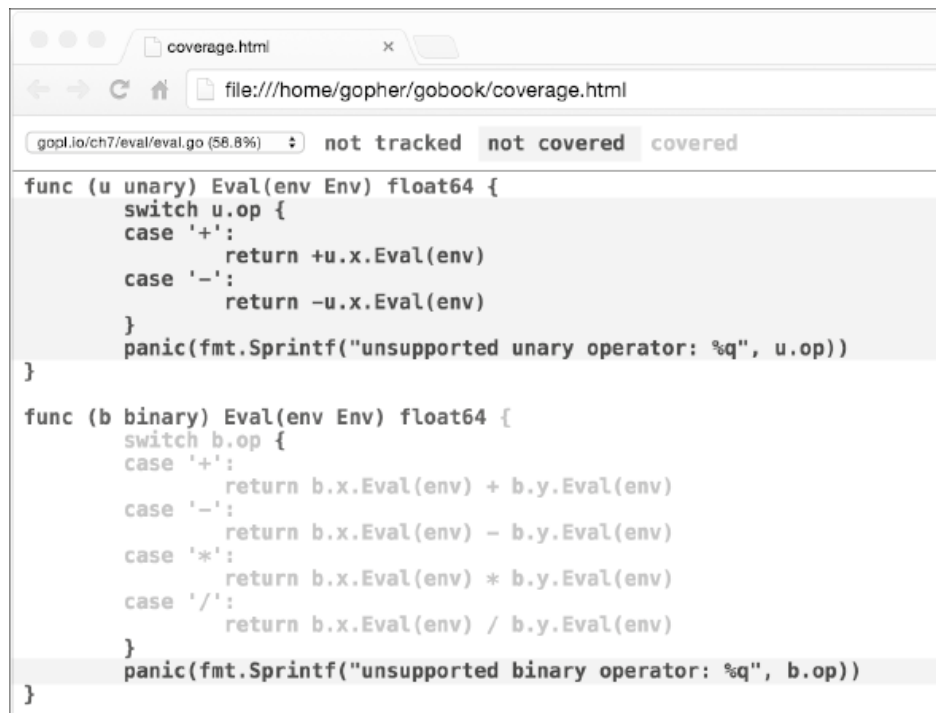


Figura 11.3 – Um relatório de cobertura.

Alcançar 100% de cobertura nas instruções soa como um objetivo nobre, mas geralmente não é viável na prática e, provavelmente, não será uma boa aplicação de esforços. O fato de uma instrução ser executada não significa que ela está livre de bugs; instruções contendo expressões complexas devem ser executadas muitas vezes, com entradas diferentes, para incluir os casos interessantes. Algumas instruções, como as instruções **panic** anteriores, não podem ser alcançadas. Outras instruções, como aquelas que tratam erros incomuns, são difíceis de exercitar, mas é raro que sejam alcançadas na prática. Testar é fundamentalmente um empreendimento pragmático: é um equilíbrio entre o custo de escrever os

testes e o custo das falhas que poderiam ser evitadas por eles. As ferramentas de cobertura podem ajudar a identificar os pontos mais fracos, porém, criar bons casos de teste exige o mesmo planejamento rigoroso que a programação em geral.

11.4 Funções Benchmark

Benchmarking é a prática de medir o desempenho de um programa com uma carga de trabalho fixa. Em Go, uma função de benchmark se parece com uma função de teste, mas tem o prefixo **Benchmark** e um parâmetro ***testing.B** que oferece a maior parte dos mesmos métodos que ***testing.T**, mais alguns métodos extras relacionados à medida de desempenho. Ela também expõe um campo inteiro **N** que especifica o número de vezes que a operação que está sendo avaliada deve ser executada.

Eis um benchmark para **IsPalindrome** que a chama **N** vezes em um loop.

```
import "testing"

func BenchmarkIsPalindrome(b *testing.B) {
    for i := 0; i < b.N; i++ {
        IsPalindrome("A man, a plan, a canal: Panama")
    }
}
```

Ela é executada com o comando a seguir. Diferente dos testes, por padrão, nenhum benchmark é executado. O argumento da flag **-bench** seleciona quais benchmarks devem ser executados. É uma expressão regular que faz a correspondência de nomes de funções **Benchmark**, cujo valor default não casa com nenhum deles. O padrão **“.”** casa todos os benchmarks do pacote **word**, mas como há apenas um, **-bench=IsPalindrome** seria equivalente.

```
$ cd $GOPATH/src/gopl.io/ch11/word2
$ go test -bench=.
PASS
BenchmarkIsPalindrome-8 1000000          1035 ns/op
ok      gopl.io/ch11/word2      2.179s
```

O sufixo numérico do nome do benchmark, **8** nesse caso, informa o valor de **GOMAXPROCS**, que é importante para benchmarks concorrentes.

O relatório informa que cada chamada a **IsPalindrome** demorou aproximadamente 1035 microssegundos, que é uma média sobre um milhão de execuções. Como a ferramenta de execução do benchmark inicialmente não tinha ideia da duração da operação, ela faz algumas medidas usando valores baixos de **N** e, então, extrapola para um valor alto o suficiente para que uma medida de tempo estável seja feita.

O motivo para o loop ser implementado pela função de benchmark, e não pelo código que faz a chamada no executor de teste, é que a função de benchmark tem a oportunidade de executar qualquer código de configuração necessário executado uma só vez fora do loop, sem que isso represente acréscimos no tempo medido para cada iteração. Se esse código de configuração ainda estiver perturbando o resultado, o parâmetro **testing.B** oferece métodos para parar, retomar e reiniciar o timer, mas raramente são necessários.

Agora que temos um benchmark e testes, é fácil testar ideias para deixar o programa mais rápido. Talvez a otimização mais óbvia seja fazer o segundo loop de **IsPalindrome** interromper verificações quando estiver no meio, a fim de evitar que efetue cada comparação duas vezes:

```
n := len(letters)/2
for i := 0; i < n; i++ {
    if letters[i] != letters[len(letters)-1-i] {
        return false
    }
}
return true
```

Porém, como ocorre com frequência, uma otimização óbvia nem sempre produz o benefício esperado. Essa otimização resultou em apenas 4% de melhoria em um experimento.

```
$ go test -bench=.
PASS
BenchmarkIsPalindrome-8 1000000          992 ns/op
ok      gopl.io/ch11/word2      2.093s
```


Outra ideia é pré-alocar um array suficientemente grande a ser usado por **letters**, em vez de expandi-lo em chamadas sucessivas a **append**. Declarar **letters** como um array do tamanho correto, assim:

```
letters := make([]rune, 0, len(s))
for _, r := range s {
    if unicode.IsLetter(r) {
        letters = append(letters, unicode.ToLower(r))
    }
}
```

resulta em uma melhoria de aproximadamente 35%, e a ferramenta de execução de benchmark agora informa a média com base em dois milhões de iterações.

```
$ go test -bench=.
PASS
BenchmarkIsPalindrome-8 2000000          697 ns/op
ok      gopl.io/ch11/word2      1.468s
```

Como mostra esse exemplo, o programa mais rápido muitas vezes é aquele que faz menos alocações de memória. A flag de linha de comando **-benchmem** fará estatísticas de alocação de memória serem incluídas em seu relatório. A seguir, comparamos o número de alocações antes da otimização:

```
$ go test -bench=. -benchmem
PASS
BenchmarkIsPalindrome    1000000  1026 ns/op   304 B/op   4 allocs/op
```

e depois dela:

```
$ go test -bench=. -benchmem
PASS
BenchmarkIsPalindrome    2000000   807 ns/op   128 B/op   1 allocs/op
```

Consolidar as alocações em uma única chamada a **make** eliminou 75% das alocações e reduziu a quantidade de memória alocada para a metade. Benchmarks como esse informam o tempo absoluto necessário para uma dada operação, mas, em muitos ambientes, as perguntas interessantes de desempenho são sobre os tempos *relativos* de duas operações diferentes. Por exemplo, se uma função demora um milissegundo para processar mil

elementos, quanto tempo ela gastará para processar dez mil ou um milhão? Essas comparações revelam o crescimento assintótico do tempo de execução de uma função. Outro exemplo: qual é o melhor tamanho para um buffer de E/S? Benchmarks do throughput de uma aplicação por uma série de tamanhos pode ajudar a escolher o menor buffer que ofereça um desempenho satisfatório. Um terceiro exemplo: qual algoritmo tem o melhor desempenho para uma dada tarefa? Benchmarks que avaliem dois algoritmos diferentes sobre os mesmos dados de entrada muitas vezes podem mostrar os pontos fortes e os pontos fracos de cada um em cargas de trabalho importantes ou representativas.

Benchmarks de comparação são apenas códigos comuns. Geralmente, eles assumem a forma de uma única função parametrizada, chamada de várias funções **Benchmark** com valores diferentes; assim:

```
func benchmark(b *testing.B, size int) { /* ... */ }
func Benchmark10(b *testing.B) { benchmark(b, 10) }
func Benchmark100(b *testing.B) { benchmark(b, 100) }
func Benchmark1000(b *testing.B) { benchmark(b, 1000) }
```

O parâmetro **size**, que especifica o tamanho da entrada, varia entre os benchmarks, mas é constante em cada benchmark. Resista à tentação de usar o parâmetro **b.N** como o tamanho da entrada. A menos que você o interprete como um contador de iterações para uma entrada de tamanho fixo, o resultado de seu benchmark não fará sentido.

Padrões revelados por benchmarks de comparação são particularmente úteis durante o design do programa, mas não devemos jogar fora os benchmarks quando o programa estiver funcionando. À medida que o programa evoluir ou sua entrada aumentar, ou se ele for implantado em novos sistemas operacionais ou processadores com características diferentes, podemos reutilizar esses benchmarks para rever decisões de design.

Exercício 11.6: Escreva benchmarks para comparar a implementação de **PopCount** da seção 2.6.2 com suas soluções dos exercícios 2.4 e 2.5. Em que ponto a abordagem orientada a tabela passa a compensar?

Exercício 11.7: Escreva benchmarks para `Add`, `UnionWith` e outros métodos de `*IntSet` (seção 6.5) usando entradas grandes pseudoaleatórias. Com que rapidez você pode fazer esses métodos executarem? Como a escolha do tamanho da palavra afeta o desempenho? Quão mais rápido é `IntSet` quando comparada a uma implementação de conjunto baseada no tipo embutido mapa?

11.5 Profiling

Benchmarks são úteis para medir o desempenho de operações específicas, mas quando tentamos fazer um programa lento ser mais rápido, muitas vezes não temos a mínima ideia de onde começar. Todo programador conhece a máxima de Donald Knuth sobre otimização prematura, que apareceu no artigo “Structured Programming with go to Statements” (Programação estruturada com instruções goto) em 1974. Embora seja com frequência erroneamente interpretado como se quisesse dizer que o desempenho não importa, em seu contexto original podemos discernir um significado diferente:

Não há dúvida de que o Graal da eficiência resulta em abuso. Programadores desperdiçam muito tempo pensando ou se preocupando com a velocidade de partes não críticas de seus programas, e essas tentativas de ser eficiente, na verdade, têm um forte impacto negativo quando a depuração e a manutenção são consideradas. *Devemos* nos esquecer as pequenas eficiências, digamos, 97% do tempo: a otimização prematura é a raiz de todo o mal.

Apesar disso, não devemos deixar passar nossas oportunidades naqueles 3% críticos. Um bom programador não será levado a ser complacente por causa dessa forma de pensar; ele será sábio, observando cuidadosamente o código crítico, mas somente *depois* que esse código tiver sido identificado. Frequentemente, é um erro fazer julgamentos *a priori* sobre quais partes de um programa são realmente críticas, pois a experiência universal dos programadores que têm

usado ferramentas de medição mostra que seus palpites intuitivos falham.³

Quando quisermos observar cuidadosamente a velocidade de nossos programas, a melhor técnica para identificar o código crítico é a *profiling* (geração de perfil). Profiling é uma abordagem automatizada para medida de desempenho baseada na amostragem de alguns *eventos* de perfil durante a execução e, em seguida, fazendo uma extrapolação desses dados em um passo após o processamento; o resumo estatístico resultante se chama *profile* (perfil).

Go suporta muitos tipos de profiling, cada qual preocupado com um aspecto diferente do desempenho, mas todos eles envolvem registrar uma sequência de eventos de interesse, cada uma com uma stack trace que a acompanha – a pilha de chamadas de funções ativas no momento do evento. A ferramenta **go test** tem suporte incluído para vários tipos de geração de perfis.

Um *perfil de CPU* (CPU profile) identifica as funções cujas execuções exigem mais tempo de CPU. A thread executando no momento em cada CPU é interrompida periodicamente pelo sistema operacional a intervalos de alguns milissegundos, com cada interrupção registrando um evento de perfil antes que a execução normal seja retomada.

Um *perfil de heap* (heap profile) identifica as instruções responsáveis por alocar mais memória. A biblioteca de geração de perfis toma amostras de chamadas a rotinas internas de alocação de memória de modo que, na média, um evento de perfil seja registrado a cada 512 KB de memória alocada.

Um *perfil de bloqueio* (blocking profile) identifica as operações responsáveis por bloquear gorrotinas por mais tempo, como chamadas de sistema, envios e recepções em canais e aquisições de locks. A biblioteca de geração de perfis registra um evento sempre que uma gorrotina é bloqueada por uma dessas operações.

Coletar um perfil para o código em teste é fácil e basta habilitar uma das

flags a seguir. Porém tome cuidado quando usar mais de uma flag ao mesmo tempo: o sistema, para coletar os dados de um tipo de perfil, pode distorcer o resultado de outros.

```
$ go test -cpuprofile=cpu.out  
$ go test -blockprofile=block.out  
$ go test -memprofile=mem.out
```

É fácil adicionar suporte à geração de perfis a programas que não sejam de teste também, embora os detalhes de como fazer isso variam entre ferramentas de linha de comando de curta duração e aplicações servidoras de longa duração. Gerar perfis é especialmente útil em aplicações de longa duração; desse modo, os recursos de geração de perfis do runtime de Go podem ser habilitados pelo programador usando a API de `runtime`.

Após termos coletado um perfil, precisamos analisá-lo usando a ferramenta `pprof`. Essa é uma parte padrão da distribuição de Go, mas como não é uma ferramenta de uso cotidiano, ela é acessada indiretamente usando `go tool pprof`. Ela tem dezenas de recursos e opções, mas o uso básico exige apenas dois argumentos: o executável que gerou o perfil e o log do perfil.

Para deixar o profiling eficiente e economizar espaço, o log não inclui nomes de funções; em seu lugar, as funções são identificadas pelos seus endereços. Isso quer dizer que `pprof` precisa do executável para compreender o log. Embora `go test` em geral descarte o executável de teste depois que o teste termina, quando o profiling está habilitado, ele salva o executável como `foo.test`, em que `foo` é o nome do pacote testado.

Os comandos a seguir mostram como coletar e exibir um perfil simples de CPU. Seleccionamos um dos benchmarks do pacote `net/http`. Normalmente é melhor gerar o perfil com benchmarks específicos construídos para serem representativos das cargas de trabalho em que alguém está interessado. Casos de teste para benchmarking quase nunca são representativos, motivo pelo qual nós os desabilitamos usando o filtro `-run=NONE`.

```
$ go test -run=NONE -bench=ClientServerParallelTLS64 \
  -cpuprofile=cpu.log net/http
PASS
BenchmarkClientServerParallelTLS64-8 1000
  3141325 ns/op 143010 B/op 1747 allocs/op
ok      net/http      3.395s

$ go tool pprof -text -nodecount=10 ./http.test cpu.log
2570ms of 3590ms total (71.59%)
Dropped 129 nodes (cum <= 17.95ms)
Showing top 10 nodes out of 166 (cum >= 60ms)
   flat  flat%  sum%   cum  cum%
1730ms  48.19%  48.19% 1750ms  48.75% crypto/elliptic.p256ReduceDegree
 230ms   6.41%  54.60%  250ms   6.96% crypto/elliptic.p256Diff
 120ms   3.34%  57.94%  120ms   3.34% math/big.addMulVVW
 110ms   3.06%  61.00%  110ms   3.06% syscall.Syscall
  90ms   2.51%  63.51% 1130ms  31.48% crypto/elliptic.p256Square
  70ms   1.95%  65.46%  120ms   3.34% runtime.scanobject
  60ms   1.67%  67.13%  830ms  23.12% crypto/elliptic.p256Mul
  60ms   1.67%  68.80%  190ms   5.29% math/big.nat.montgomery
  50ms   1.39%  70.19%   50ms   1.39% crypto/elliptic.p256ReduceCarry
  50ms   1.39%  71.59%   60ms   1.67% crypto/elliptic.p256Sum
```

A flag **-text** especifica o formato da saída; nesse caso, é uma tabela textual, com uma linha por função, ordenada de modo que as funções “mais quentes” – aquelas que consomem mais ciclos de CPU – aparecem antes. A flag **-nodecount=10** limita o resultado em dez linhas. Para problemas mais evidentes de desempenho, esse formato textual pode ser suficiente para identificar a causa.

Esse perfil informa que a criptografia de curva elíptica (elliptic-curve cryptography) é importante para o desempenho desse benchmark HTTPS em particular. Por outro lado, se um perfil estiver dominado por funções de alocação de memória do pacote **runtime**, reduzir o consumo de memória pode ser uma otimização que valha a pena fazer.

Para problemas mais sutis, talvez seja melhor usar uma das apresentações gráficas de **pprof**. Elas exigem o GraphViz, que pode ser baixado a partir de www.graphviz.org. A flag **-web** então apresenta um grafo direcionado das funções do programa, com anotações contendo os números de seus perfis de CPU e coloridos para indicar as funções mais quentes.

Mal tocamos a superfície das ferramentas de geração de perfis de Go aqui. Para saber mais, leia o artigo “Profiling Go Programs” (Gerando

perfis de programas Go) em Go Blog.

11.6 Funções Example

O terceiro tipo de função tratada de forma especial por **go test** é uma função de exemplo: uma função cujo nome começa com **Example**. Ela não tem nem parâmetros nem resultados. Eis uma função de exemplo para **IsPalindrome**:

```
func ExampleIsPalindrome() {  
    fmt.Println(IsPalindrome("A man, a plan, a canal: Panama"))  
    fmt.Println(IsPalindrome("palindrome"))  
    // Output:  
    // true  
    // false  
}
```

Funções de exemplo têm três propósitos. O principal é servir de documentação: um bom exemplo pode ser uma maneira mais sucinta e intuitiva de informar o comportamento de uma função de biblioteca do que uma descrição textual, especialmente quando usada como um lembrete ou como uma referência rápida. Um exemplo também pode demonstrar a interação entre vários tipos e funções que pertencem a uma API, enquanto uma documentação textual sempre deve estar associada a um lugar, por exemplo, uma declaração de tipo ou de função ou ao pacote como um todo. Diferentemente de exemplos em comentários, as funções de exemplo são códigos Go verdadeiros, sujeitos a verificações em tempo de compilação; portanto, elas não se tornam ultrapassadas à medida que o código evolui.

Com base no sufixo da função **Example**, o servidor de documentação **godoc** baseado em web associa funções de exemplo à função ou ao pacote que elas exemplificam, portanto, **ExampleIsPalindrome** seria mostrada com a documentação da função **IsPalindrome**, e uma função de exemplo chamada simplesmente de **Example** seria associada ao pacote **word** como um todo.

O segundo propósito é que os exemplos são testes que podem ser

executados por **go test**. Se a função de exemplo tiver um comentário **// Output:** no final, como mostrado anteriormente, o executor de testes invocará a função e verificará se o que seria exibido na saída-padrão coincide com o texto no comentário.

O terceiro propósito de um exemplo é a experimentação mão na massa. O servidor **godoc** em **golang.org** usa o Go Playground para deixar o usuário editar e executar cada função de exemplo de dentro de um navegador web, como mostra a figura 11.4. Com frequência, essa é a maneira mais rápida de ter uma ideia do que é uma função ou um recurso da linguagem em particular.

Os dois últimos capítulos do livro analisam os pacotes **reflect** e **unsafe**, que poucos programadores Go usam em geral – e menos programadores ainda *precisam* usar. Se você ainda não escreveu nenhum programa substancial em Go, agora é uma boa hora para fazer isso.



Figura 11.4 – Um exemplo interativo de strings.Join em godoc.

1 N.T.: Tradução livre de acordo com a citação original em inglês.

2 Nota do Revisor da Tradução: semi-palíndromo é uma palavra que, invertida, produz outra palavra existente. Exemplos: desserts e stressed, arroz e zorra.

3 N.T.: Tradução livre feita de acordo com a citação original em inglês.

12

Reflexão

Go oferece um mecanismo para atualizar variáveis e inspecionar seus valores em tempo de execução, chamar seus métodos e aplicar as operações intrínsecas a suas representações, tudo sem conhecer seus tipos em tempo de compilação. Esse sistema se chama *reflexão* (reflection). A reflexão também permite tratar os próprios tipos como valores de primeira classe.

Neste capítulo, exploraremos os recursos de reflexão de Go para ver como eles aumentam a expressividade da linguagem e, em particular, como são fundamentais para a implementação de duas APIs importantes: formatação de string, oferecida por `fmt`, e codificação de protocolos, oferecida por pacotes como `encoding/json` e `encoding/xml`. A reflexão também é essencial para o mecanismo de templates oferecido pelos pacotes `text/template` e `html/template` que vimos na seção 4.6. Entretanto, é difícil raciocinar sobre reflexão, e ela só deve ser usada com cuidado. Então, embora esses pacotes sejam implementados usando reflexão, eles não a expõem em suas próprias APIs.

12.1 Por que usar reflexão?

Às vezes, precisamos escrever uma função capaz de lidar uniformemente com valores de tipos que não satisfazem uma interface comum, não têm uma representação conhecida ou não existem no momento em que projetamos a função – ou com essas três características.

Um exemplo familiar é a lógica de formatação em `fmt.Fprintf`, que em geral é capaz de exibir um valor arbitrário de qualquer tipo, até mesmo de um tipo definido pelo usuário. Vamos tentar implementar uma função como ela usando o que já sabemos. Por questões de simplicidade, nossa

função aceitará um argumento e devolverá o resultado na forma de uma string, como faz `fmt.Sprint`; portanto, nós a chamaremos de `Sprint`.

Começaremos com um switch de tipo que testa se o argumento define um método `String`, que será chamada, em caso afirmativo. Então, acrescentamos casos ao switch que testam o tipo dinâmico do valor em relação a cada um dos tipos básicos – `string`, `int`, `bool`, e assim por diante – e executam a operação apropriada de formatação em cada caso.

```
func Sprint(x interface{}) string {
    type stringer interface {
        String() string
    }
    switch x := x.(type) {
    case stringer:
        return x.String()
    case string:
        return x
    case int:
        return strconv.Itoa(x)
    // ...casos similares para int16, uint32 e assim por diante...
    case bool:
        if x {
            return "true"
        }
        return "false"
    default:
        // array, chan, func, map, pointer, slice, struct
        return "???"
    }
}
```

Mas como lidamos com tipos como `[]float64`, `map[string][]string` e outros? Poderíamos acrescentar mais casos, mas o número de tipos como esses é infinito. E o que dizer de tipos nomeados, como `url.Values`? Mesmo que o switch de tipo tivesse um caso para seu tipo subjacente `map[string][]string`, ele não corresponderia a `url.Values`, porque os dois tipos não são idênticos, e o switch de tipo não pode incluir um caso para cada tipo como `url.Values`, pois isso exigiria que essa biblioteca dependesse de seus clientes.

Sem uma maneira de inspecionar a representação dos valores de tipos desconhecidos, rapidamente ficamos sem ação. O que precisamos é de reflexão.

12.2 `reflect.Type` e `reflect.Value`

A reflexão é oferecida pelo pacote `reflect`. Esse pacote define dois tipos importantes: `Type` e `Value`. Um `Type` representa um tipo de Go. É uma interface com vários métodos para distinguir entre tipos e inspecionar seus componentes, como os campos de uma estrutura ou os parâmetros de uma função. A única implementação de `reflect.Type` é o descritor de tipo (seção 7.5): a mesma entidade que identifica o tipo dinâmico de um valor interface.

A função `reflect.TypeOf` aceita qualquer `interface{}` e devolve seu tipo dinâmico como um `reflect.Type`:

```
t := reflect.TypeOf(3)    // um reflect.Type
fmt.Println(t.String())   // "int"
fmt.Println(t)            // "int"
```

A chamada anterior a `TypeOf(3)` atribui o valor `3` ao parâmetro `interface{}`. Lembre-se da seção 7.5 que uma atribuição de um valor concreto para um tipo interface realiza uma conversão implícita de interface que cria um valor interface constituído de dois componentes: seu *tipo dinâmico* é o tipo do operando (`int`), e seu *valor dinâmico* é o valor do operando (`3`).

Como `reflect.TypeOf` devolve o tipo dinâmico de um valor interface, ele sempre devolve um tipo concreto. Por exemplo, o código a seguir exibe `"*os.File"`, e não `"io.Writer"`. Mais adiante, veremos que `reflect.Type` é capaz de representar tipos interface também.

```
var w io.Writer = os.Stdout
fmt.Println(reflect.TypeOf(w)) // "*os.File"
```

Observe que `reflect.Type` satisfaz `fmt.Stringer`. Como exibir o tipo dinâmico de um valor interface é útil para depuração e logging, `fmt.Printf` oferece uma forma concisa `%T` que usa `reflect.TypeOf`

internamente:

```
fmt.Printf("%T\n", 3) // "int"
```

O outro tipo importante no pacote `reflect` é `Value`. Um `reflect.Value` pode armazenar um valor de qualquer tipo. A função `reflect.ValueOf` aceita qualquer `interface{}` e devolve um `reflect.Value` contendo o valor dinâmico da interface. Como ocorre com `reflect.TypeOf`, os resultados de `reflect.ValueOf` são sempre concretos, mas um `reflect.Value` pode armazenar valores interface também.

```
v := reflect.ValueOf(3) // um reflect.Value
fmt.Println(v)          // "3"
fmt.Printf("%v\n", v)   // "3"
fmt.Println(v.String()) // NOTA: "<int Value>"
```

Assim como `reflect.Type`, `reflect.Value` também satisfaz `fmt.Stringer`, mas a menos que `Value` armazene uma string, o resultado do método `String` revela apenas o tipo. Em seu lugar, usamos o verbo `%v` do pacote `fmt`, que trata `reflect.Values` de modo especial.

Chamar o método `Type` em um `Value` devolve seu tipo como um `reflect.Type`:

```
t := v.Type() // um reflect.Type
fmt.Println(t.String()) // "int"
```

A operação inversa de `reflect.ValueOf` é o método `reflect.Value.Interface`. Esse método devolve uma `interface{}` que armazena o mesmo valor concreto que `reflect.Value`:

```
v := reflect.ValueOf(3) // um reflect.Value
x := v.Interface()      // uma interface{}
i := x.(int)            // um int
fmt.Printf("%d\n", i)   // "3"
```

Um `reflect.Value` e uma `interface{}` podem ambos armazenar qualquer valor. A diferença é que uma interface vazia oculta a representação e as operações intrínsecas do valor que ela armazena e não expõe nenhum de seus métodos, portanto, a menos que saibamos qual é seu tipo dinâmico e usemos uma asserção de tipo para espiar dentro dela (como fizemos anteriormente), há pouca coisa que podemos fazer com o

valor que ela contém. Em comparação, um **Value** tem muitos métodos para inspecionar seu conteúdo, independentemente de seu tipo. Vamos usá-los em nossa segunda tentativa de uma função genérica de formatação, que chamaremos de **format.Any**.

Em vez de usar um switch de tipo, utilizaremos o método **Kind** de **reflect.Value** para diferenciar os casos. Embora haja infinitamente muitos tipos, há apenas um número finito de *tipos* de tipo: os tipos básicos **Bool**, **String** e todos os números; os tipos agregados **Array** e **Struct**; os tipos referência **Chan**, **Func**, **Ptr**, **Slice** e **Map**; os tipos **Interface**; e, por fim, **Invalid**, que significa nenhum valor. (O valor zero de um **reflect.Value** tem o tipo **Invalid**.)

gopl.io/ch12/format

```
package format

import (
    "reflect"
    "strconv"
)

// Any formata qualquer valor como uma string.
func Any(value interface{}) string {
    return formatAtom(reflect.ValueOf(value))
}

// formatAtom formata um valor sem inspecionar sua estrutura interna.
func formatAtom(v reflect.Value) string {
    switch v.Kind() {
    case reflect.Invalid:
        return "invalid"
    case reflect.Int, reflect.Int8, reflect.Int16,
        reflect.Int32, reflect.Int64:
        return strconv.FormatInt(v.Int(), 10)
    case reflect.Uint, reflect.Uint8, reflect.Uint16,
        reflect.Uint32, reflect.Uint64, reflect.Uintptr:
        return strconv.FormatUint(v.Uint(), 10)
    // ...casos para números de ponto flutuante e complexos foram omitidos
    // por concisão...
    case reflect.Bool:
        return strconv.FormatBool(v.Bool())
    case reflect.String:
```

```

        return strconv.Quote(v.String())
    case reflect.Chan, reflect.Func, reflect.Ptr, reflect.Slice, reflect.Map:
        return v.Type().String() + " 0x" +
            strconv.FormatUint(uint64(v.Pointer()), 16)
    default: // reflect.Array, reflect.Struct, reflect.Interface
        return v.Type().String() + " value"
    }
}

```

Até agora, nossa função trata cada valor como algo indivisível, sem estrutura interna – daí o nome **formatAtom**. Para tipos agregados (estruturas e arrays) e interfaces, ela exibe apenas o *tipo* do valor, enquanto para tipos referência (canais, funções, ponteiros, fatias e mapas), ela exibe o tipo e o endereço da referência em hexadecimal. Não é o ideal, mas é uma grande melhoria; como **Kind** está interessado somente na representação subjacente, **format.Any** também funciona para tipos nomeados. Por exemplo:

```

var x int64 = 1
var d time.Duration = 1 * time.Nanosecond
fmt.Println(format.Any(x))           // "1"
fmt.Println(format.Any(d))           // "1"
fmt.Println(format.Any([]int64{x}))  // "[int64 0x8202b87b0]"
fmt.Println(format.Any([]time.Duration{d})) // "[time.Duration 0x8202b87e0]"

```

12.3 Display: um printer recursivo de valores

Em seguida, veremos como melhorar a apresentação de tipos compostos. Em vez de tentar copiar **fmt.Sprint** exatamente, criaremos uma função utilitária de depuração chamada **Display** que, dado um valor complexo **x** qualquer, exibe a estrutura completa desse valor, rotulando cada elemento com o caminho pelo qual ele foi encontrado. Vamos começar com um exemplo.

```

e, _ := eval.Parse("sqrt(A / pi)")
Display("e", e)

```

Na chamada anterior, o argumento de **Display** é uma árvore sintática do avaliador de expressões da seção 7.9. A saída de **Display** está mostrada a seguir:

```

Display e (eval.call):
e.fn = "sqrt"
e.args[0].type = eval.binary
e.args[0].value.op = 47
e.args[0].value.x.type = eval.Var
e.args[0].value.x.value = "A"
e.args[0].value.y.type = eval.Var
e.args[0].value.y.value = "pi"

```

Sempre que possível, evite expor a reflexão na API de um pacote. Definiremos uma função **display** não exportada para fazer o verdadeiro trabalho de recursão e exportaremos **Display**, que é um wrapper simples em torno dela, o qual aceita um parâmetro `interface{}`:

gopl.io/ch12/display

```

func Display(name string, x interface{}) {
    fmt.Printf("Display %s (%T):\n", name, x)
    display(name, reflect.ValueOf(x))
}

```

Em **display**, utilizaremos a função **formatAtom** já definida para exibir valores elementares – tipos básicos, funções e canais –, mas usaremos os métodos de **reflect.Value** para exibir recursivamente cada componente de um tipo mais complexo. Conforme a recursão desce, a string **path**, que a princípio descreve caminho inicial (por exemplo, **"e"**), será expandida para indicar como alcançamos o valor atual (por exemplo, **"e.args[0].value"**).

Como já não estamos fingindo implementar **fmt.Sprint**, usaremos o pacote **fmt** para deixar nosso exemplo conciso.

```

func display(path string, v reflect.Value) {
    switch v.Kind() {
    case reflect.Invalid:
        fmt.Printf("%s = invalid\n", path)
    case reflect.Slice, reflect.Array:
        for i := 0; i < v.Len(); i++ {
            display(fmt.Sprintf("%s[%d]", path, i), v.Index(i))
        }
    case reflect.Struct:
        for i := 0; i < v.NumField(); i++ {
            fieldPath := fmt.Sprintf("%s.%s", path, v.Type().Field(i).Name)

```



```

        display(fieldPath, v.Field(i))
    }
    case reflect.Map:
        for _, key := range v.MapKeys() {
            display(fmt.Sprintf("%s[%s]", path,
                formatAtom(key)), v.MapIndex(key))
        }
    case reflect.Ptr:
        if v.IsNil() {
            fmt.Printf("%s = nil\n", path)
        } else {
            display(fmt.Sprintf("(%s)", path), v.Elem())
        }
    case reflect.Interface:
        if v.IsNil() {
            fmt.Printf("%s = nil\n", path)
        } else {
            fmt.Printf("%s.type = %s\n", path, v.Elem().Type())
            display(path+".value", v.Elem())
        }
    default: // tipos básicos, canais, funções
        fmt.Printf("%s = %s\n", path, formatAtom(v))
    }
}

```

Vamos discutir os casos na sequência.

Fatias e arrays: a lógica é a mesma para ambos. O método **Len** devolve o número de elementos de um valor do tipo fatia ou array, e **Index(i)** recupera o elemento no índice **i**, também como um **reflect.Value**; um pânico é gerado se **i** estiver fora dos limites. São análogos às operações embutidas **len(a)** e **a[i]** de sequências. A função **display** chama a si mesma recursivamente em cada elemento da sequência, concatenando a notação de índice "[i]" ao caminho.

Embora **reflect.Value** tenha muitos métodos, apenas alguns são seguros para chamar em qualquer dado valor. Por exemplo, o método **Index** pode ser chamado em valores do tipo **Slice**, **Array** ou **String**, mas, para qualquer outro tipo, gera pânico.

Estruturas: o método **NumField** informa o número de campos da estrutura,

e `Field(i)` devolve o valor do i -ésimo campo como um `reflect.Value`. A lista de campos inclui aqueles promovidos a partir de campos anônimos. Para acrescentar a notação de seletor de campo `".f"` no caminho, devemos obter o `reflect.Type` da estrutura e acessar o nome de seu i -ésimo campo.

Mapas: o método `MapKeys` devolve uma fatia de `reflect.Values`, um para cada chave do mapa. Como geralmente acontece quando iteramos por um mapa, a ordem é indefinida. `MapIndex(key)` devolve o valor correspondente a `key`. Concatenamos a notação de indexação `"[key]"` ao caminho. (Estamos fazendo uma simplificação aqui. O tipo de uma chave de mapa não está restrito aos tipos que `formatAtom` trata; arrays, estruturas e interfaces também podem ser chaves válidas de mapa. Estender esse caso para exibir a chave de forma completa será feito no exercício 12.1.)

Ponteiros: o método `Elem` devolve a variável apontada por um ponteiro, novamente como um `reflect.Value`. Essa operação seria segura, mesmo se o valor do ponteiro fosse `nil`, caso em que o resultado seria do tipo `Invalid`, mas usamos `IsNil` para identificar ponteiros nil explicitamente e poder exibir uma mensagem mais apropriada. Prefixamos o caminho com um `"*"` e o colocamos entre parênteses para evitar ambiguidade.

Interfaces: novamente, usamos `IsNil` para testar se a interface é nil e, se não for, recuperamos seu valor dinâmico usando `v.Elem()` e exibimos seu tipo e seu valor.

Agora que nossa função `Display` está completa, vamos colocá-la para trabalhar. O tipo `Movie` a seguir é uma pequena variação da versão que está na seção 4.5:

```
type Movie struct {
    Title, Subtitle string
    Year             int
    Color            bool
    Actor            map[string]string
    Oscars           []string
    Sequel           *string
}
```

```
}
```

Vamos declarar um valor desse tipo e ver o que **Display** faz com ele:

```
strangelove := Movie{
  Title:      "Dr. Strangelove",
  Subtitle:   "How I Learned to Stop Worrying and Love the Bomb",
  Year:       1964,
  Color:      false,
  Actor: map[string]string{
    "Dr. Strangelove":      "Peter Sellers",
    "Grp. Capt. Lionel Mandrake": "Peter Sellers",
    "Pres. Merkin Muffley":  "Peter Sellers",
    "Gen. Buck Turgidson":   "George C. Scott",
    "Brig. Gen. Jack D. Ripper": "Sterling Hayden",
    `Maj. T.J. "King" Kong`:  "Slim Pickens",
  },
  Oscars: []string{
    "Best Actor (Nomin.)",
    "Best Adapted Screenplay (Nomin.)",
    "Best Director (Nomin.)",
    "Best Picture (Nomin.)",
  },
}
```

A chamada a **Display("strangelove", strangelove)** exhibe:

```
Display strangelove (display.Movie):
strangelove.Title = "Dr. Strangelove"
strangelove.Subtitle = "How I Learned to Stop Worrying and Love the Bomb"
strangelove.Year = 1964
strangelove.Color = false
strangelove.Actor["Gen. Buck Turgidson"] = "George C. Scott"
strangelove.Actor["Brig. Gen. Jack D. Ripper"] = "Sterling Hayden"
strangelove.Actor["Maj. T.J. \"King\" Kong"] = "Slim Pickens"
strangelove.Actor["Dr. Strangelove"] = "Peter Sellers"
strangelove.Actor["Grp. Capt. Lionel Mandrake"] = "Peter Sellers"
strangelove.Actor["Pres. Merkin Muffley"] = "Peter Sellers"
strangelove.Oscars[0] = "Best Actor (Nomin.)"
strangelove.Oscars[1] = "Best Adapted Screenplay (Nomin.)"
strangelove.Oscars[2] = "Best Director (Nomin.)"
strangelove.Oscars[3] = "Best Picture (Nomin.)"
strangelove.Sequel = nil
```

Podemos usar **Display** para exibir a parte interna dos tipos da biblioteca,

como `*os.File`:

```
Display("os.Stderr", os.Stderr)
// Output:
// Display os.Stderr (*os.File):
// (*(os.Stderr).file).fd = 2
// (*(os.Stderr).file).name = "/dev/stderr"
// (*(os.Stderr).file).npipe = 0
```

Observe que até mesmo campos não exportados são visíveis à reflexão. Preste atenção, pois a saída em particular desse exemplo pode variar entre plataformas e mudar com o tempo, à medida que as bibliotecas evoluem. (Há um motivo para esses campos serem privados!) Podemos até mesmo aplicar `Display` em um `reflect.Value` e observá-lo percorrer a representação interna do descritor de tipos para `*os.File`. A saída da chamada a `Display("rV", reflect.ValueOf(os.Stderr))` está a seguir, embora, é claro, você possa ter um resultado diferente:

```
Display rV (reflect.Value):
(*rV.typ).size = 8
(*rV.typ).hash = 871609668
(*rV.typ).align = 8
(*rV.typ).fieldAlign = 8
(*rV.typ).kind = 22
(*(*rV.typ).string) = "*os.File"
(*(*(*rV.typ).uncommonType).methods[0].name) = "Chdir"
(*(*(*(*rV.typ).uncommonType).methods[0].mtyp).string) = "func() error"
(*(*(*(*rV.typ).uncommonType).methods[0].typ).string) = "func(*os.File) error"
...
```

Observe a diferença entre estes dois exemplos:

```
var i interface{} = 3
Display("i", i)
// Output:
// Display i (int):
// i = 3
Display("&i", &i)
// Output:
// Display &i (*interface {}):
// (*&i).type = int
// (*&i).value = 3
```

No primeiro exemplo, **Display** chama `reflect.ValueOf(i)`, que devolve um valor do tipo `Int`. Como mencionamos na seção 12.2, `reflect.ValueOf` sempre devolve um `Value` de um tipo concreto, pois ele extrai o conteúdo de um valor interface.

No segundo exemplo, **Display** chama `reflect.ValueOf(&i)`, que devolve um ponteiro para `i` do tipo `Ptr`. O caso do switch para `Ptr` chama `Elem` nesse valor, que devolve um `Value` representando a própria *variável* `i`, do tipo `Interface`. Um `Value` obtido indiretamente, como esse, pode representar qualquer valor, incluindo interfaces. A função **display** chama a si mesma recursivamente e, dessa vez, exhibe componentes separados para o tipo e o valor dinâmicos da interface.

Como implementado no momento, **Display** jamais terminará se encontrar um ciclo no grafo de objetos, como esta lista ligada que aponta para sua própria cauda:

```
// uma estrutura que aponta para si mesma
type Cycle struct{ Value int; Tail *Cycle }
var c Cycle
c = Cycle{42, &c}
Display("c", c)
```

Display exhibe esta expansão eternamente crescente:

```
Display c (display.Cycle):
c.Value = 42
(*c.Tail).Value = 42
(*(*c.Tail).Tail).Value = 42
(*(*(*c.Tail).Tail).Tail).Value = 42
...ad infinitum...
```

Muitos programas Go contêm pelo menos alguns dados cíclicos. Tornar **Display** robusto contra esses ciclos é complicado, exigindo controles adicionais para registrar o conjunto de referências que foram seguidas até agora, o que também é custoso. Uma solução genérica exige recursos da linguagem de **unsafe**, que veremos na seção 13.3.

Os ciclos representam menos problemas para `fmt.Sprint` porque ela raramente tenta exibir a estrutura completa. Por exemplo, quando encontra um ponteiro, ela quebra a recursão exibindo o valor numérico

do ponteiro. Ela pode ficar travada tentando exibir uma fatia ou um mapa que contenha a si mesmo como um elemento, mas casos raros como esse não justificam o trabalho adicional considerável de tratar ciclos.

Exercício 12.1: Estenda `Display` para que ela possa exibir mapas cujas chaves sejam estruturas ou arrays.

Exercício 12.2: Torne `display` segura para ser usada em estruturas de dados cíclicas limitando o número de passos que ela executa antes de abandonar a recursão. (Na seção 13.3, veremos outra maneira de identificar ciclos.)

12.4 Exemplo: Codificando expressões-S

`Display` é uma rotina de depuração para exibir dados estruturados, mas não está tão longe de ser capaz de codificar ou de fazer *marshaling* de objetos Go quaisquer como mensagens em uma notação portátil adequada para comunicação entre processos.

Como vimos na seção 4.5, a biblioteca-padrão de Go aceita vários formatos, incluindo JSON, XML e ASN.1. Outra notação ainda amplamente usada são as *expressões-S* (S-expressions), da sintaxe de Lisp. Diferente de outras notações, as expressões-S não são tratadas pela biblioteca-padrão de Go, principalmente por não terem uma definição universalmente aceita, apesar de diversas tentativas de padronização e da existência de muitas implementações.

Nesta seção, definiremos um pacote que codifica objetos Go quaisquer usando uma notação de expressão-S que aceite as seguintes construções:

42	integer
"hello"	string (com aspas em estilo Go)
foo	symbol (um nome sem aspas)
(1 2 3)	list (zero ou mais itens entre parênteses)

Booleanos são tradicionalmente codificados com o símbolo `t` para verdadeiro (true), e a lista vazia `()` ou o símbolo `nil` para falso, mas, por questões de simplicidade, nossa implementação irá ignorá-los, assim como os canais e funções, pois seus estados são opacos para a reflexão. Ela ignorará também números de ponto flutuante reais e complexos, além

de interfaces. Acrescentar suporte a eles será feito no exercício 12.3.

Codificaremos os tipos de Go usando expressões-S como explicado a seguir. Inteiros e strings são codificados da maneira óbvia. Valores nil são codificados como o símbolo **nil**. Arrays e fatias são codificados usando a notação de lista.

Estruturas são codificadas como uma lista de associações de campos, em que cada associação é uma lista de dois elementos, dos quais o primeiro (um símbolo) é o nome do campo e o segundo é o valor do campo. Os mapas também são codificados como uma lista de pares, em que cada par é composto da chave e do valor de uma entrada do mapa. Tradicionalmente, expressões-S representam listas de pares chave/valor usando uma única célula *cons* (**key . value**) para cada par, em vez de usar uma lista de dois elementos, mas, para simplificar a decodificação, vamos ignorar a notação de lista com ponto.

A codificação é feita por uma única função recursiva **encode**, mostrada a seguir. Sua estrutura é essencialmente a mesma de **Display** da seção anterior:

gopl.io/ch12/sexpr

```
func encode(buf *bytes.Buffer, v reflect.Value) error {
    switch v.Kind() {
    case reflect.Invalid:
        buf.WriteString("nil")

    case reflect.Int, reflect.Int8, reflect.Int16,
        reflect.Int32, reflect.Int64:
        fmt.Fprintf(buf, "%d", v.Int())

    case reflect.Uint, reflect.Uint8, reflect.Uint16,
        reflect.Uint32, reflect.Uint64, reflect.Uintptr:
        fmt.Fprintf(buf, "%d", v.Uint())

    case reflect.String:
        fmt.Fprintf(buf, "%q", v.String())

    case reflect.Ptr:
        return encode(buf, v.Elem())

    case reflect.Array, reflect.Slice: // (valor ...)
        buf.WriteByte('(')
```

```

    for i := 0; i < v.Len(); i++ {
        if i > 0 {
            buf.WriteByte(' ')
        }
        if err := encode(buf, v.Index(i)); err != nil {
            return err
        }
    }
    buf.WriteByte(')')

case reflect.Struct: // ((nome valor) ...)
    buf.WriteByte('(')
    for i := 0; i < v.NumField(); i++ {
        if i > 0 {
            buf.WriteByte(' ')
        }
        fmt.Fprintf(buf, "(%s ", v.Type().Field(i).Name)
        if err := encode(buf, v.Field(i)); err != nil {
            return err
        }
        buf.WriteByte(')')
    }
    buf.WriteByte(')')

case reflect.Map: // ((chave valor) ...)
    buf.WriteByte('(')
    for i, key := range v.MapKeys() {
        if i > 0 {
            buf.WriteByte(' ')
        }
        buf.WriteByte('(')
        if err := encode(buf, key); err != nil {
            return err
        }
        buf.WriteByte(' ')
        if err := encode(buf, v.MapIndex(key)); err != nil {
            return err
        }
        buf.WriteByte(')')
    }
    buf.WriteByte(')')

default: // float, complex, bool, chan, func, interface
    return fmt.Errorf("unsupported type: %s", v.Type())
}

```



```
    return nil
}
```

A função **Marshal** encapsula o codificador em uma API semelhante às de outros pacotes **encoding/...**:

```
// Marshal codifica um valor Go em um formato de expressão-S.
func Marshal(v interface{}) ([]byte, error) {
    var buf bytes.Buffer
    if err := encode(&buf, reflect.ValueOf(v)); err != nil {
        return nil, err
    }
    return buf.Bytes(), nil
}
```

Eis a saída de **Marshal** aplicada à variável **strangelove** da seção 12.3:

```
((Title "Dr. Strangelove") (Subtitle "How I Learned to Stop Worrying and Love the Bomb") (Year 1964) (Actor (("Grp. Capt. Lionel Mandrake" "Peter Sellers") ("Pres. Merkin Muffley" "Peter Sellers") ("Gen. Buck Turgidson" "George C. Scott") ("Brig. Gen. Jack D. Ripper" "Sterling Hayden") ("Maj. T.J. \King\" Kong" "Slim Pickens") ("Dr. Strangelove" "Peter Sellers")))) (Oscars ("Best Actor (Nomin.)" "Best Adapted Screenplay (Nomin.)" "Best Director (Nomin.)" "Best Picture (Nomin.)")) (Sequel nil))
```

A saída toda é formada com uma linha longa com espaços mínimos, dificultando sua leitura. A seguir, apresentamos a mesma saída formatada manualmente de acordo com as convenções de expressões-S. Escrever um pretty-printer para expressões-S fica como um exercício (desafiador); o download de **gopl.io** inclui uma versão simples.

```
((Title "Dr. Strangelove")
 (Subtitle "How I Learned to Stop Worrying and Love the Bomb")
 (Year 1964)
 (Actor (("Grp. Capt. Lionel Mandrake" "Peter Sellers")
          ("Pres. Merkin Muffley" "Peter Sellers")
          ("Gen. Buck Turgidson" "George C. Scott")
          ("Brig. Gen. Jack D. Ripper" "Sterling Hayden")
          ("Maj. T.J. \King\" Kong" "Slim Pickens")
          ("Dr. Strangelove" "Peter Sellers"))))
 (Oscars ("Best Actor (Nomin.)"
          "Best Adapted Screenplay (Nomin.)"
          "Best Director (Nomin.)"
          "Best Picture (Nomin.)"))
 (Sequel nil))
```

Assim como as funções `fmt.Print`, `json.Marshal` e `Display`, `sexpr.Marshal` terá um loop infinito se for chamada com dados cíclicos.

Na seção 12.6, esboçaremos a implementação da função correspondente de decodificação de expressões-S, mas antes de chegar lá precisamos entender como a reflexão pode ser usada para atualizar variáveis.

Exercício 12.3: Implemente os casos da função `encode` que estão faltando. Codifique booleanos como `t` e `nil`, números de ponto flutuante usando a notação de Go e números complexos como `1+2i` como `#C(1.0 2.0)`. As interfaces podem ser codificadas como um par contendo um nome de tipo e um valor, por exemplo, `("[]int" (1 2 3))`, mas tome cuidado, pois essa notação é ambígua: o método `reflect.Type.String` pode devolver a mesma string para tipos diferentes.

Exercício 12.4: Modifique `encode` para fazer um pretty-print de expressões-S no estilo mostrado anteriormente.

Exercício 12.5: Adapte `encode` para gerar JSON no lugar de expressões-S. Teste seu codificador usando o decodificador padrão `json.Unmarshal`.

Exercício 12.6: Adapte `encode` para que, como uma otimização, ele não codifique um campo cujo valor seja o valor zero de seu tipo.

Exercício 12.7: Crie uma API de streaming para o codificador de expressões-S seguindo o estilo de `json.Encoder` (seção 4.5).

12.5 Atualizando variáveis com `reflect.Value`

Até agora, a reflexão apenas *interpretava* valores em nosso programa de diversas maneiras. O ponto principal desta seção, porém, é *modificá-los*.

Lembre-se de que algumas expressões em Go, como `x`, `x.f[1]` e `*p`, representam variáveis, mas outras como `x + 1` e `f(2)` não representam. Uma variável é uma área de armazenamento *endereçável* que contém um valor, e seu valor pode ser atualizado por meio desse endereço.

Uma distinção semelhante se aplica a `reflect.Values`. Alguns são endereçáveis, outros não. Considere as declarações a seguir:

	// valor	tipo	variável?
<code>x := 2</code>			
<code>a := reflect.ValueOf(2)</code>	// 2	int	não
<code>b := reflect.ValueOf(x)</code>	// 2	int	não
<code>c := reflect.ValueOf(&x)</code>	// &x	*int	não
<code>d := c.Elem()</code>	// 2	int	sim (x)

O valor em **a** não é endereçável. É simplesmente uma cópia do inteiro 2. O mesmo vale para **b**. O valor em **c** também não é endereçável, pois é uma cópia do valor de ponteiro **&x**. De fato, nenhum **reflect.Value** devolvido por **reflect.ValueOf(x)** é endereçável. Porém **d**, derivado de **c** por desreferência do ponteiro nele contido, referencia uma variável e, assim, é endereçável. Podemos usar essa abordagem chamando **reflect.ValueOf(&x).Elem()** para obter um **Value** endereçável para qualquer variável **x**.

Podemos perguntar se um **reflect.Value** é endereçável por meio de seu método **CanAddr**:

```
fmt.Println(a.CanAddr()) // "false"
fmt.Println(b.CanAddr()) // "false"
fmt.Println(c.CanAddr()) // "false"
fmt.Println(d.CanAddr()) // "true"
```

Obtemos um **reflect.Value** endereçável sempre que fizermos um acesso indireto por meio de um ponteiro, mesmo que comecemos por um **Value** não endereçável. Todas as regras usuais de possibilidade de endereçamento têm regras análogas para reflexão. Por exemplo, como a expressão de indexação de fatia **e[i]** segue implicitamente um ponteiro, ela é endereçável, mesmo que a expressão **e** não seja. Por analogia, **reflect.ValueOf(e).Index(i)** refere-se a uma variável **e**, desse modo, é endereçável, mesmo que **reflect.ValueOf(e)** não seja.

Recuperar a variável a partir de um **reflect.Value** endereçável exige três passos. Em primeiro lugar, chamamos **Addr()**; ela devolve um **Value** que armazena um ponteiro para a variável. Em seguida, chamamos **Interface()** nesse **Value**, que devolve um valor **interface{}** contendo o ponteiro. Por fim, se conhecemos o tipo da variável, podemos usar uma asserção de tipo para recuperar o conteúdo da interface como um ponteiro comum. Podemos, então, atualizar a variável por meio do

ponteiro:

```
x := 2
d := reflect.ValueOf(&x).Elem() // d refere-se à variável x
px := d.Addr().Interface().(*int) // px := &x
*px = 3                          // x = 3
fmt.Println(x)                  // "3"
```

Ou podemos atualizar a variável referenciada por um **reflect.Value** endereçável diretamente, sem usar um ponteiro, chamando o método **reflect.Value.Set**:

```
d.Set(reflect.ValueOf(4))
fmt.Println(x) // "4"
```

As mesmas verificações para possibilidade de atribuição, comumente realizadas pelo compilador, são feitas em tempo de execução pelos métodos **Set**. No exemplo anterior, tanto a variável quanto o valor têm tipo **int**, mas se a variável fosse um **int64**, o programa geraria pânico, portanto, é muito importante garantir que o valor possa ser atribuído ao tipo da variável:

```
d.Set(reflect.ValueOf(int64(5))) //pânico: int64 não pode ser atribuído a int
```

E é claro que chamar **Set** em um **reflect.Value** não endereçável também gera pânico:

```
x := 2
b := reflect.ValueOf(x)
b.Set(reflect.ValueOf(3)) // pânico: Set usando um valor não endereçável
```

Existem variantes de **Set** especializadas para determinados grupos de tipos básicos: **SetInt**, **SetUint**, **SetString**, **SetFloat**, e assim por diante:

```
d := reflect.ValueOf(&x).Elem()
d.SetInt(3)
fmt.Println(x) // "3"
```

Em certos aspectos, esses métodos são mais tolerantes. **SetInt**, por exemplo, será bem-sucedido, desde que o tipo da variável seja algum tipo de inteiro com sinal ou até mesmo um tipo nomeado cujo tipo subjacente seja um inteiro com sinal; se o valor for grande demais, ele será silenciosamente truncado para que caiba. Contudo seja cauteloso:

chamar `SetInt` em um `reflect.Value` que se refira a uma variável `interface{}` gerará pânico, mesmo que `Set` seja bem-sucedido.

```
x := 1
rx := reflect.ValueOf(&x).Elem()
rx.SetInt(2)           // OK, x = 2
rx.Set(reflect.ValueOf(3)) // OK, x = 3
rx.SetString("hello")   // pânico: string não pode ser atribuída a int
rx.Set(reflect.ValueOf("hello")) // pânico: string não pode ser atribuída a int

var y interface{}
ry := reflect.ValueOf(&y).Elem()
ry.SetInt(2)           // pânico: SetInt chamado em um Value de interface
ry.Set(reflect.ValueOf(3)) // OK, y = int(3)
ry.SetString("hello")   // pânico: SetString chamado em um Value de interface
ry.Set(reflect.ValueOf("hello")) // OK, y = "hello"
```

Quando aplicamos `Display` em `os.Stdout`, descobrimos que a reflexão pode ler valores de campos não exportados de estruturas que são inacessíveis de acordo com as regras usuais da linguagem, como o campo `fd int` de uma estrutura `os.File` em uma plataforma compatível com Unix. No entanto, a reflexão não pode atualizar esses valores:

```
stdout := reflect.ValueOf(os.Stdout).Elem() // *os.Stdout, uma variável
                                              // os.File
fmt.Println(stdout.Type())                  // "os.File"
fd := stdout.FieldByName("fd")
fmt.Println(fd.Int()) // "1"
fd.SetInt(2)          // pânico: campo não exportado
```

Um `reflect.Value` endereçável registra se ele foi obtido percorrendo um campo de estrutura não exportado e, em caso afirmativo, não permite modificação. Como consequência, `CanAddr` geralmente não é a verificação correta a ser usada antes de atualizar uma variável. O método relacionado `CanSet` informa se um `reflect.Value` é endereçável e se pode ser atualizado:

```
fmt.Println(fd.CanAddr(), fd.CanSet()) // "true false"
```

12.6 Exemplo: Decodificando expressões-S

Para cada função **Marshal** oferecida pelos pacotes **encoding/...** da biblioteca-padrão, há uma função **Unmarshal** correspondente que faz a decodificação. Por exemplo, como vimos na seção 4.5, dada uma fatia de bytes contendo dados codificados em JSON para nosso tipo **Movie** (seção 12.3), podemos decodificá-la assim:

```
data := []byte{/* ... */}
var movie Movie
err := json.Unmarshal(data, &movie)
```

A função **Unmarshal** usa reflexão para modificar os campos da variável **movie** existente, criando novos mapas, estruturas e fatias, conforme determinado pelo tipo **Movie** e pelo conteúdo dos dados de entrada.

Vamos agora implementar uma função **Unmarshal** simples para expressões-S, análoga à função **json.Unmarshal** padrão usada antes, que seja a inversa de nossa **sexpr.Marshal**. Observe que uma implementação robusta e genérica exige mais código do que caberá confortavelmente nesse exemplo, que já é longo; assim, tomamos muitos atalhos. Tratamos apenas um subconjunto limitado de expressões-S e não tratamos erros com elegância. O código pretende ilustrar a reflexão, e não o parsing.

O analisador léxico (lexer) usa o tipo **Scanner** do pacote **text/scanner** para separar um stream de entrada em uma sequência de tokens, como comentários, identificadores, strings literais e números literais. O método **Scan** de scanner faz o scanner avançar e devolve o tipo do próximo token, que tem o tipo **rune**. A maioria dos tokens, como '(', é constituída de uma única runa, mas o pacote **text/scanner** representa os tipos dos tokens de múltiplos caracteres **Ident**, **String** e **Int**, usando valores negativos baixos do tipo **rune**. Após uma chamada a **Scan** que devolve um desses tipos de token, o método **TokenText** do scanner devolve o texto do token.

Como um parser típico pode precisar inspecionar o token atual várias vezes, mas o método **Scan** faz o scanner avançar, encapsulamos o scanner em um tipo auxiliar chamado **lexer**, que guarda o token mais recentemente devolvido por **Scan**.

gopl.io/ch12/sexpr

```
type lexer struct {
    scan scanner.Scanner
    token rune // o token atual
}

func (lex *lexer) next()          { lex.token = lex.scan.Scan() }
func (lex *lexer) text() string { return lex.scan.TokenText() }

func (lex *lexer) consume(want rune) {
    if lex.token != want { // NOTA: não é um exemplo de um bom tratamento
                           // de erro.
        panic(fmt.Sprintf("got %q, want %q", lex.text(), want))
    }
    lex.next()
}
```

Vamos agora nos voltar para o parser. Ele é constituído de duas funções principais. A primeira delas, **read**, lê a expressão-S que começa no token atual e atualiza a variável referenciada pelo **reflect.Value** **v** endereçável.

```
func read(lex *lexer, v reflect.Value) {
    switch lex.token {
    case scanner.Ident:
        // Os únicos identificadores válidos são
        // "nil" e os nomes dos campos de estrutura.
        if lex.text() == "nil" {
            v.Set(reflect.Zero(v.Type()))
            lex.next()
            return
        }
    case scanner.String:
        s, _ := strconv.Unquote(lex.text()) // NOTA: ignorando erros
        v.SetString(s)
        lex.next()
        return
    case scanner.Int:
        i, _ := strconv.Atoi(lex.text()) // NOTA: ignorando erros
        v.SetInt(int64(i))
        lex.next()
        return
    case '(':
        lex.next()
    }
```

```

    readList(lex, v)
    lex.next() // consome ')'
    return
}
panic(fmt.Sprintf("unexpected token %q", lex.text()))
}

```

Nossas expressões-S usam identificadores para dois propósitos distintos: nomes de campos de estrutura e o valor `nil` para um ponteiro. A função `read` só trata o último caso. Quando encontra o `scanner.Ident "nil"`, ela define `v` com o valor zero de seu tipo usando a função `reflect.Zero`. Para qualquer outro identificador, um erro é informado. A função `readList`, que veremos em breve, trata identificadores usados como nomes de campos de estrutura.

Um token `'('` indica o início de uma lista. A segunda função, `readList`, decodifica uma lista em uma variável de tipo composto – um mapa, uma estrutura, uma fatia ou um array – de acordo com o tipo de variável Go que estamos preenchendo no momento. Em cada caso, o loop continua fazendo parsing dos itens até encontrar o parêntese de fechamento correspondente, `)'`, conforme identificado pela função `endList`.

A parte interessante está na recursão. O caso mais simples é um array. Até o `)'` de fechamento ser visto, usamos `Index` para obter a variável para cada elemento do array e fazemos uma chamada recursiva para `read`, a fim de preenchê-la. Como em muitos outros casos de erro, se os dados de entrada fizerem o decodificador indexar além do final do array, ela gera um pânico. Uma abordagem semelhante é usada para fatias, exceto que devemos criar uma nova variável para cada elemento, preenchê-la e, então, adicioná-la à fatia.

Os loops para estruturas e mapas devem fazer parse de uma sublista (**key value**) em cada iteração. Para estruturas, a chave é um símbolo que identifica o campo. Análogo ao caso dos arrays, adquirimos a variável existente para o campo da estrutura usando `FieldByName` e fazemos uma chamada recursiva para preenchê-la. Para mapas, a chave pode ser de qualquer tipo e, de modo análogo ao caso das fatias, criamos uma nova variável, a preenchemos recursivamente e, por fim, inserimos o novo par

chave/valor no mapa.

```
func readList(lex *lexer, v reflect.Value) {
    switch v.Kind() {
    case reflect.Array: // (item ...)
        for i := 0; !endList(lex); i++ {
            read(lex, v.Index(i))
        }
    case reflect.Slice: // (item ...)
        for !endList(lex) {
            item := reflect.New(v.Type().Elem()).Elem()
            read(lex, item)
            v.Set(reflect.Append(v, item))
        }
    case reflect.Struct: // ((nome valor) ...)
        for !endList(lex) {
            lex.consume('(')
            if lex.token != scanner.Ident {
                panic(fmt.Sprintf("got token %q, want field name", lex.text()))
            }
            name := lex.text()
            lex.next()
            read(lex, v.FieldByName(name))
            lex.consume(')')
        }
    case reflect.Map: // ((chave valor) ...)
        v.Set(reflect.MakeMap(v.Type()))
        for !endList(lex) {
            lex.consume('(')
            key := reflect.New(v.Type().Key()).Elem()
            read(lex, key)
            value := reflect.New(v.Type().Elem()).Elem()
            read(lex, value)
            v.SetMapIndex(key, value)
            lex.consume(')')
        }
    default:
        panic(fmt.Sprintf("cannot decode list into %v", v.Type()))
    }
}

func endList(lex *lexer) bool {
```

```

switch lex.token {
case scanner.EOF:
    panic("end of file")
case ')':
    return true
}
return false
}

```

Por fim, encapsulamos o parser em uma função exportada `Unmarshal`, mostrada a seguir, que oculta algumas das arestas da implementação. Erros encontrados durante o parsing resultam em pânico, portanto, `Unmarshal` usa uma chamada adiada para se recuperar do pânico (seção 5.10) e devolve uma mensagem de erro.

```

// Unmarshal faz parse dos dados de uma expressão-S e preenche a variável
// cujo endereço está no ponteiro não nil out.
func Unmarshal(data []byte, out interface{}) (err error) {
    lex := &lexer{scan: scanner.Scanner{Mode: scanner.GoTokens}}
    lex.scan.Init(bytes.NewReader(data))
    lex.next() // obtém o primeiro token
    defer func() {
        // NOTA: este não é um exemplo de um tratamento de erro ideal.
        if x := recover(); x != nil {
            err = fmt.Errorf("error at %s: %v", lex.scan.Position, x)
        }
    }()
    read(lex, reflect.ValueOf(out).Elem())
    return nil
}

```

Uma implementação com qualidade para ambiente de produção não deve gerar pânico para nenhuma entrada e deve oferecer um erro informativo para cada contratempo, talvez com um número de linha ou um offset. Apesar disso, esperamos que esse exemplo dê uma ideia do que acontece internamente nos pacotes como `encoding/json` e como você pode usar reflexão para preencher estruturas de dados.

Exercício 12.8: A função `sexpr.Unmarshal`, assim como `json.Unmarshal`, exige a entrada completa em uma fatia de bytes antes de poder iniciar a decodificação. Defina um tipo `sexpr.Decoder` que, como `json.Decoder`,

permita que uma sequência de valores seja decodificada a partir de um `io.Reader`. Mude `sexpr.Unmarshal` para que use esse novo tipo.

Exercício 12.9: Escreva uma API baseada em token para decodificar expressões-S, seguindo o estilo de `xml.Decoder` (seção 7.14). Você precisará dos cinco tipos de tokens: `Symbol`, `String`, `Int`, `StartList` e `EndList`.

Exercício 12.10: Estenda `sexpr.Unmarshal` para que trate booleanos, números de ponto flutuante e interfaces codificados pela sua solução no exercício 12.3. (Dica: para decodificar interfaces, você precisará de um mapeamento do nome de cada tipo tratado para seu `reflect.Type`.)

12.7 Acessando tags de campos de estrutura

Na seção 4.5, usamos *tags de campos* (field tags) de estrutura para modificar a codificação JSON de valores de estruturas em Go. A tag de campo `json` permite escolher nomes alternativos de campo e suprimir a saída de campos vazios. Nesta seção, veremos como acessar tags de campo usando reflexão.

Em um servidor web, a primeira tarefa da maioria das funções de handler HTTP é extrair os parâmetros da requisição em variáveis locais. Definiremos uma função utilitária `params.Unpack` que usa tags de campos de estrutura para deixar a escrita de handlers HTTP (seção 7.7) mais conveniente.

Inicialmente, mostraremos como usá-la. A função `search` a seguir é um handler HTTP. Ela define uma variável chamada `data` de um tipo anônimo de estrutura cujos campos correspondem aos parâmetros da requisição HTTP. As tags de campo da estrutura especificam os nomes dos parâmetros, que com frequência são curtos e enigmáticos, pois o espaço é precioso em um URL. A função `Unpack` preenche a estrutura a partir da requisição, de modo que os parâmetros possam ser acessados de forma conveniente e com um tipo apropriado.

gopl.io/ch12/search

```
import "gopl.io/ch12/params"
```

```
// search implementa o endpoint do URL /search.
func search(resp http.ResponseWriter, req *http.Request) {
    var data struct {
        Labels []string `http:"l"`
        MaxResults int `http:"max"`
        Exact bool `http:"x"`
    }
    data.MaxResults = 10 // set default
    if err := params.Unpack(req, &data); err != nil {
        http.Error(resp, err.Error(), http.StatusBadRequest) // 400
        return
    }
    // ...restante do handler...
    fmt.Fprintf(resp, "Search: %v\n", data)
}
```

A função **Unpack** a seguir faz três tarefas. Em primeiro lugar, ela chama **req.ParseForm()** para fazer parse da requisição. Em seguida, **req.Form** contém todos os parâmetros, não importando se o cliente HTTP usou o método de requisição GET ou POST.

Por fim, **Unpack** cria um mapeamento do nome *efetivo* de cada campo para a variável deste. O nome efetivo pode ser diferente do nome verdadeiro se o campo tiver uma tag. O método **Field** de **reflect.Type** devolve um **reflect.StructField** que fornece informações sobre o tipo de cada campo, como seu nome, o tipo e a tag opcional. O campo **Tag** é um **reflect.StructTag** que é do tipo string e oferece um método **Get** para fazer parse e extrair a substring para uma chave em particular, como **http:"..."** nesse caso.

gopl.io/ch12/params

```
// Unpack preenche os campos da estrutura apontada por ptr
// a partir dos parâmetros da requisição HTTP em req.
func Unpack(req *http.Request, ptr interface{}) error {
    if err := req.ParseForm(); err != nil {
        return err
    }

    // Cria um mapa de campos cujas chaves são o nome efetivo.
    fields := make(map[string]reflect.Value)
    v := reflect.ValueOf(ptr).Elem() // a variável da estrutura
    for i := 0; i < v.NumField(); i++ {
```

```

    fieldInfo := v.Type().Field(i) // um reflect.StructField
    tag := fieldInfo.Tag           // um reflect.StructTag
    name := tag.Get("http")
    if name == "" {
        name = strings.ToLower(fieldInfo.Name)
    }
    fields[name] = v.Field(i)
}

// Atualiza o campo da estrutura para cada parâmetro da requisição.
for name, values := range req.Form {
    f := fields[name]
    if !f.IsValid() {
        continue // ignora parâmetros HTTP não reconhecidos
    }
    for _, value := range values {
        if f.Kind() == reflect.Slice {
            elem := reflect.New(f.Type().Elem()).Elem()
            if err := populate(elem, value); err != nil {
                return fmt.Errorf("%s: %v", name, err)
            }
            f.Set(reflect.Append(f, elem))
        } else {
            if err := populate(f, value); err != nil {
                return fmt.Errorf("%s: %v", name, err)
            }
        }
    }
}
return nil
}

```

Por fim, **Unpack** itera pelos pares nome/valor dos parâmetros HTTP e atualiza os campos correspondentes da estrutura. Lembre-se de que o mesmo nome de parâmetro pode aparecer mais de uma vez. Se isso acontecer e o campo for uma fatia, todos os valores desse parâmetro serão acumulados na fatia. Caso contrário, o campo será repetidamente sobrescrito de modo que apenas o último valor terá efeito.

A função **populate** cuida de definir um único campo **v** (ou um único elemento de um campo fatia) de um valor de parâmetro. Por enquanto, ela aceita apenas strings, inteiros com sinal e booleanos. Tratar outros tipos

fica como exercício.

```
func populate(v reflect.Value, value string) error {
    switch v.Kind() {
    case reflect.String:
        v.SetString(value)
    case reflect.Int:
        i, err := strconv.ParseInt(value, 10, 64)
        if err != nil {
            return err
        }
        v.SetInt(i)
    case reflect.Bool:
        b, err := strconv.ParseBool(value)
        if err != nil {
            return err
        }
        v.SetBool(b)
    default:
        return fmt.Errorf("unsupported kind %s", v.Type())
    }
    return nil
}
```

Se acrescentarmos o handler **server** em um servidor web, esta poderia ser uma sessão típica:

```
$ go build gopl.io/ch12/search
$ ./search &
$ ./fetch 'http://localhost:12345/search'
Search: {Labels:[] MaxResults:10 Exact:false}
$ ./fetch 'http://localhost:12345/search?l=golang&l=programming'
Search: {Labels:[golang programming] MaxResults:10 Exact:false}
$ ./fetch 'http://localhost:12345/search?l=golang&l=programming&max=100'
Search: {Labels:[golang programming] MaxResults:100 Exact:false}
$ ./fetch 'http://localhost:12345/search?x=true&l=golang&l=programming'
Search: {Labels:[golang programming] MaxResults:10 Exact:true}
$ ./fetch 'http://localhost:12345/search?q=hello&x=123'
x: strconv.ParseBool: parsing "123": invalid syntax
$ ./fetch 'http://localhost:12345/search?q=hello&max=lots'
max: strconv.ParseInt: parsing "lots": invalid syntax
```

Exercício 12.11: Escreva a função **Pack** correspondente. Dado um valor de estrutura, **Pack** deve devolver um URL que incorpore os valores de

parâmetro da estrutura.

Exercício 12.12: Estenda a notação de tag de campos para expressar requisitos de validade de parâmetros. Por exemplo, pode ser que uma string precise ser um endereço de email válido ou um número de cartão de crédito, e um inteiro precise ser um CEP válido nos Estados Unidos. Modifique **Unpack** para verificar esses requisitos.

Exercício 12.13: Modifique o codificador (seção 12.4) e o decodificador (seção 12.6) de expressões-S para que eles considerem a tag de campo **sexpr: "..."** de modo semelhante a **encoding/json** (seção 4.5).

12.8 Exibindo os métodos de um tipo

Nosso último exemplo de reflexão usa **reflect.Type** para exibir o tipo de um valor arbitrário e listar seus métodos:

gopl.io/ch12/methods

```
// Print exibe o conjunto de métodos do valor x.
func Print(x interface{}) {
    v := reflect.ValueOf(x)
    t := v.Type()
    fmt.Printf("type %s\n", t)

    for i := 0; i < v.NumMethod(); i++ {
        methType := v.Method(i).Type()
        fmt.Printf("func (%s) %s%s\n", t, t.Method(i).Name,
            strings.TrimPrefix(methType.String(), "func"))
    }
}
```

Tanto **reflect.Type** quanto **reflect.Value** têm um método chamado **Method**. Cada chamada a **t.Method(i)** devolve uma instância de **reflect.Method**, um tipo estrutura que descreve o nome e o tipo de um único método. Cada chamada a **v.Method(i)** devolve um **reflect.Value** que representa o valor de método (seção 6.4), isto é, um método vinculado ao seu receptor. Usando o método **reflect.Value.Call** (não temos espaço para mostrá-lo aqui), é possível chamar **Values** do tipo **Func** como esse, porém, esse programa só precisa de seu **Type**.

Eis os métodos que pertencem a dois tipos: `time.Duration` e `*strings.Replacer`:

```
methods.Print(time.Hour)
// Output:
// type time.Duration
// func (time.Duration) Hours() float64
// func (time.Duration) Minutes() float64
// func (time.Duration) Nanoseconds() int64
// func (time.Duration) Seconds() float64
// func (time.Duration) String() string

methods.Print(new(strings.Replacer))
// Output:
// type *strings.Replacer
// func (*strings.Replacer) Replace(string) string
// func (*strings.Replacer) WriteString(io.Writer, string) (int, error)
```

12.9 Uma advertência

Há muito mais a dizer sobre a API de reflexão, mas não temos espaço aqui para mostrar; no entanto, os exemplos anteriores deram uma ideia do que é possível fazer. A reflexão é uma ferramenta poderosa e expressiva, mas deve ser usada com cuidado por três motivos.

O primeiro é que um código baseado em reflexão pode ser frágil. Para todo erro que faria um compilador informar um erro de tipo, há uma maneira correspondente de usar erroneamente a reflexão, mas enquanto o compilador informa o erro em tempo de build, um erro de reflexão é informado durante a execução na forma de pânico, possivelmente muito tempo depois de o programa ter sido escrito ou até mesmo muito tempo depois de ter iniciado sua execução.

Se a função `readList` (seção 12.6), por exemplo, tivesse que ler uma string da entrada enquanto preenchesse uma variável do tipo `int`, a chamada a `reflect.Value.SetString` geraria pânico. A maioria dos programas que usa reflexão tem perigos semelhantes, e um cuidado considerável é necessário para monitorar o tipo, a capacidade de endereçamento e a possibilidade de atribuição a cada `reflect.Value`.

A melhor maneira de evitar essa fragilidade é garantir que o uso de reflexão esteja totalmente encapsulado em seu pacote e, se possível, evitar **reflect.Value** em favor de tipos específicos na API de seu pacote, a fim de restringir as entradas a valores permitidos. Se isso não for possível, execute verificações dinâmicas adicionais antes de cada operação arriscada. Como exemplo da biblioteca-padrão, quando **fmt.Printf** aplica um verbo a um operando inapropriado, ele não gera pânico misteriosamente, mas exibe uma mensagem de erro informativa. O programa continua com um bug, mas é mais fácil diagnosticá-lo.

```
fmt.Printf("%d %s\n", "hello", 42) // "%!d(string=hello) %!s(int=42)"
```

A reflexão também reduz a segurança e a precisão de refatorações automatizadas e de ferramentas de análise porque elas não podem determinar ou contar com informações de tipo.

O segundo motivo para evitar reflexão é que, como os tipos servem como uma forma de documentação e as operações de reflexão não podem estar sujeitas à verificação estática de tipo, códigos com uso intensivo de reflexão muitas vezes são difíceis de entender. Sempre documente cuidadosamente os tipos esperados e outras invariantes de funções que aceitem uma **interface{}** ou um **reflect.Value**.

O terceiro motivo é que funções baseadas em reflexão podem ser mais lentas em uma ou duas ordens de grandeza do que um código especializado para um tipo em particular. Em um programa típico, a maioria das funções não é relevante para o desempenho geral, portanto, não há problema em usar reflexão quando isso deixa o programa mais claro. Testes são um caso particularmente bom para usar reflexão, pois a maioria dos testes usa conjuntos pequenos de dados. Contudo, para funções no caminho crítico, é melhor evitar a reflexão.

13

Programação de baixo nível

O design de Go garante várias propriedades de segurança que limitam os modos como um programa Go pode “dar errado”. Durante a compilação, a verificação de tipos detecta a maioria das tentativas de aplicar uma operação a um valor que seja inadequada ao seu tipo, por exemplo, subtrair uma string de outra. Regras rígidas de conversões de tipo evitam acesso direto aos dados internos de tipos embutidos como strings, mapas, fatias e canais.

Para erros que não podem ser estaticamente identificados, como acessos fora dos limites em array ou desreferências de ponteiros nil, verificações dinâmicas garantem que o programa termine imediatamente com um erro informativo sempre que uma operação proibida ocorre. Gerenciamento automático de memória (coleta de lixo, ou garbage collection) elimina bugs do tipo “usar depois de liberar”, assim como a maior parte dos vazamentos de memória (memory leaks).

Muitos detalhes de implementação são inacessíveis aos programas Go. Não há maneiras de descobrir o layout de memória de um tipo agregado como uma estrutura, ou o código de máquina de uma função, ou a identidade da thread do sistema operacional em que a gorrotina atual está executando. Na verdade, o escalonador de Go transfere livremente as gorrotinas de uma thread para outra. Um ponteiro identifica uma variável sem revelar seu endereço numérico. Endereços podem mudar à medida que o coletor de lixo move as variáveis; ponteiros são atualizados de modo transparente.

Juntos, esses recursos deixam os programas Go, especialmente aqueles que falham, mais previsíveis e menos misteriosos que programas em C, que é a linguagem de baixo nível mais tradicional. Ao ocultar os detalhes

subjacentes, eles também deixam os programas Go altamente portáveis, pois a semântica da linguagem é, em grande medida, independente de qualquer compilador, sistema operacional ou arquitetura de CPU em particular. (Não totalmente independente: alguns detalhes passam, como o tamanho de palavra do processador, a ordem de avaliação de determinadas expressões e o conjunto de restrições de implementação imposto pelo compilador.)

Ocasionalmente, podemos optar por deixar de lado algumas dessas garantias úteis para atingir o melhor desempenho possível, interagir com bibliotecas escritas em outras linguagens ou implementar uma função que não pode ser expressa em Go puro.

Neste capítulo, veremos como o pacote **unsafe** permite deixar de lado as regras usuais e como utilizar a ferramenta **cgo** para criar vínculos (bindings) de Go com bibliotecas C e chamadas do sistema operacional.

As abordagens descritas neste capítulo não devem ser usadas de modo leviano. Sem muita atenção aos detalhes, elas podem causar os tipos de falhas imprevisíveis, misteriosas e não locais que são infelizmente conhecidas de programadores C. O uso de **unsafe** também acaba com a garantia de compatibilidade de Go com versões futuras, pois, seja de maneira proposital, seja acidental, é fácil depender de detalhes não especificados de implementação que podem mudar inesperadamente.

O pacote **unsafe** é bem mágico. Embora pareça um pacote normal e seja importado da maneira usual, na verdade, ele é implementado pelo compilador. Esse pacote oferece acesso a alguns dos recursos embutidos da linguagem que não estão comumente disponíveis porque expõem detalhes do layout de memória de Go. Apresentar esses recursos como um pacote separado deixam as raras ocasiões em que eles são necessários mais perceptíveis. Além disso, alguns ambientes podem restringir o uso do pacote **unsafe** por questões de segurança.

O pacote **unsafe** é amplamente usado em pacotes de baixo nível como **runtime**, **os**, **syscall** e **net**, que interagem com o sistema operacional, mas quase nunca é necessário em programas comuns.

13.1 `unsafe.Sizeof`, `Alignof` e `Offsetof`

A função `unsafe.Sizeof` informa o tamanho em bytes da representação de seu operando, que pode ser uma expressão de qualquer tipo; a expressão não é avaliada. Uma chamada a `Sizeof` é uma expressão constante do tipo `uintptr`, portanto, o resultado pode ser usado como a dimensão de um tipo array ou para calcular outras constantes.

```
import "unsafe"

fmt.Println(unsafe.Sizeof(float64(0))) // "8"
```

`Sizeof` informa apenas o tamanho da parte fixa de cada estrutura de dados, como o ponteiro e o tamanho de uma string, mas não partes indiretas como o conteúdo da string. Tamanhos típicos para todos os tipos não agregados de Go estão mostrados a seguir, embora os tamanhos exatos possam variar de acordo com o toolchain (cadeia de ferramentas). Por questões de portabilidade, mostramos os tamanhos dos tipos referência (ou tipos contendo referências) em termos de palavras, em que uma palavra tem 4 bytes em uma plataforma de 32 bits e 8 bytes em uma plataforma de 64 bits.

Computadores carregam e armazenam valores da memória de forma mais eficiente quando esses valores estão devidamente *alinhados*. Por exemplo, o endereço de um valor de um tipo com dois bytes, por exemplo, um `int16`, deve ser um número par, o endereço de um valor de quatro bytes como `rune` deve ser um múltiplo de quatro e o endereço de um valor de oito bytes como `float64`, `uint64` ou um ponteiro de 64 bits deve ser múltiplo de oito. Requisitos de alinhamento de múltiplos maiores são incomuns, mesmo para tipos maiores de dados, como `complex128`.

Por esse motivo, o tamanho de um valor de um tipo agregado (uma estrutura ou um array) tem pelo menos a soma dos tamanhos de seus campos ou elementos, mas pode ser maior por causa da presença de “buracos”. Os buracos são espaços não usados acrescentados pelo compilador para garantir que o campo ou o elemento seguinte esteja devidamente alinhado em relação ao início da estrutura ou do array.

Tipo	Tamanho
<code>bool</code>	1 byte
<code>int_N</code> , <code>uint_N</code> , <code>float_N</code> , <code>complex_N</code>	$N/8$ bytes (por exemplo, <code>float64</code> tem 8 bytes)
<code>int</code> , <code>uint</code> , <code>uintptr</code>	1 palavra
<code>*T</code>	1 palavra
<code>string</code>	2 palavras (data, len)
<code>[]T</code>	3 palavras (data, len, cap)
<code>map</code>	1 palavra
<code>func</code>	1 palavra
<code>chan</code>	1 palavra
<code>interface</code>	2 palavras (type, value)

A especificação da linguagem não garante que a ordem em que os campos são declarados seja a mesma em que eles são dispostos na memória; portanto, teoricamente, um compilador está livre para reorganizá-los, embora atualmente, enquanto escrevemos este livro, nenhum deles o faça. Se os tipos dos campos de uma estrutura forem de tamanhos diferentes, talvez seja mais eficiente no que diz respeito à memória declarar os campos em uma ordem que os empacote o mais concisamente possível. As três estruturas a seguir têm os mesmos campos, mas a primeira exige até 50% a mais de memória que as outras duas:

```

// 64 bits      32 bits
struct{ bool; float64; int16 } // 3 palavras 4 palavras
struct{ float64; int16; bool } // 2 palavras 3 palavras
struct{ bool; int16; float64 } // 2 palavras 3 palavras

```

Os detalhes do algoritmo de alinhamento estão além do escopo deste livro e, certamente, não vale a pena se preocupar com todas as estruturas, mas um empacotamento eficiente pode deixar estruturas de dados

alocadas com frequência mais compactas e, desse modo, mais rápidas.

A função **unsafe.Alignof** informa o alinhamento exigido para o tipo de seu argumento. Como **Sizeof**, ela pode ser aplicada a uma expressão de qualquer tipo e produz uma constante. Em geral, tipos booleanos e numéricos são alinhados para seus tamanhos (até um máximo de 8 bytes), e todos os demais tipos são alinhados por palavra.

A função **unsafe.Offsetof**, cujo operando deve ser um seletor de campo **x.f**, calcula o offset do campo **f** em relação ao início da estrutura **x** que o engloba, levando em consideração os buracos, se houver.

A figura 13.1 mostra uma variável de estrutura **x** e seu layout de memória em implementações Go típicas de 32 bits e de 64 bits. As regiões em cinza são buracos.

```
var x struct {  
    a bool  
    b int16  
    c []int  
}
```

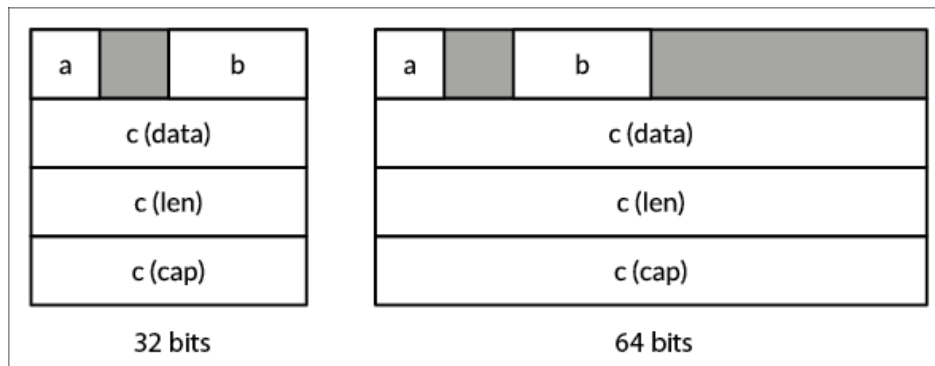


Figura 13.1 – Buracos em uma estrutura.

A tabela a seguir mostra os resultados da aplicação das três funções de **unsafe** ao próprio **x** e a cada um de seus três campos:

Plataforma típica de 32 bits:

```
Sizeof(x)   = 16  Alignof(x)   = 4  
Sizeof(x.a) = 1   Alignof(x.a) = 1  Offsetof(x.a) = 0  
Sizeof(x.b) = 2   Alignof(x.b) = 2  Offsetof(x.b) = 2  
Sizeof(x.c) = 12  Alignof(x.c) = 4  Offsetof(x.c) = 4
```

Plataforma típica de 64 bits:

```
Sizeof(x)   = 32  Alignof(x)   = 8
Sizeof(x.a) = 1   Alignof(x.a) = 1  Offsetof(x.a) = 0
Sizeof(x.b) = 2   Alignof(x.b) = 2  Offsetof(x.b) = 2
Sizeof(x.c) = 24  Alignof(x.c) = 8  Offsetof(x.c) = 8
```

Apesar de seus nomes, essas funções, na verdade, são seguras e podem ser úteis para entender o layout da memória bruta em um programa quando fizermos otimizações de espaço.

13.2 `unsafe.Pointer`

A maioria dos tipos ponteiro é escrita como `*T`, que quer dizer “um ponteiro para uma variável do tipo `T`”. O tipo `unsafe.Pointer` é um tipo especial de ponteiro capaz de armazenar o endereço de qualquer variável. É claro que não podemos fazer um acesso indireto por meio de um `unsafe.Pointer` usando `*p` porque não sabemos qual tipo essa expressão deve ter. Assim, como os ponteiros comuns, `unsafe.Pointers` são comparáveis e podem ser comparados com `nil`, que é o valor zero do tipo. Um ponteiro `*T` comum pode ser convertido em um `unsafe.Pointer`, e um `unsafe.Pointer` pode ser convertido de volta para um ponteiro comum, não necessariamente do mesmo tipo `*T`. Ao converter um ponteiro `*float64` para um `*uint64`, por exemplo, podemos inspecionar o padrão de bits de uma variável contendo um número de ponto flutuante:

```
package math

func Float64bits(f float64) uint64 { return *(*uint64)(unsafe.Pointer(&f)) }

fmt.Printf("%#016x\n", Float64bits(1.0)) // "0x3ff0000000000000"
```

Por meio do ponteiro resultante, podemos atualizar o padrão de bits também. Isso é inofensivo para uma variável de ponto flutuante, pois qualquer padrão de bits é permitido, mas, em geral, conversões de `unsafe.Pointer` nos permitem escrever valores arbitrários na memória e, desse modo, subverter o sistema de tipos.

Um `unsafe.Pointer` também pode ser convertido em um `uintptr` que armazena o valor numérico do ponteiro, o que nos permite realizar

operações aritméticas em endereços. (Lembre-se de que, de acordo com o capítulo 3, um `uintptr` é um inteiro sem sinal, grande o suficiente para representar um endereço.) Essa conversão também pode ser aplicada ao inverso, mas, novamente, converter de um `uintptr` para um `unsafe.Pointer` pode subverter o sistema de tipos, pois nem todos os números são endereços válidos.

Muitos valores de `unsafe.Pointer` são, portanto, intermediários para converter ponteiros comuns em endereços numéricos puros e vice-versa. O exemplo a seguir toma o endereço da variável `x`, soma o offset de seu campo `b`, converte o endereço resultante em `*int16` e, por meio desse ponteiro, atualiza `x.b`:

gopl.io/ch13/unsafeptr

```
var x struct {
    a bool
    b int16
    c []int
}

// equivalente a pb := &x.b
pb := (*int16)(unsafe.Pointer(
    uintptr(unsafe.Pointer(&x)) + unsafe.Offsetof(x.b)))
*pb = 42

fmt.Println(x.b) // "42"
```

Embora a sintaxe seja desajeitada – talvez não seja ruim, pois esses recursos devem ser usados esporadicamente –, não fique tentado a introduzir variáveis temporárias do tipo `uintptr` para separar as linhas. O código a seguir está incorreto:

```
// NOTA: sutilmente incorreto!
tmp := uintptr(unsafe.Pointer(&x)) + unsafe.Offsetof(x.b)
pb := (*int16)(unsafe.Pointer(tmp))
*pb = 42
```

O motivo é bem sutil. Alguns coletores de lixo movem variáveis na memória para reduzir a fragmentação ou o trabalho de gerenciamento. Coletores de lixo desse tipo são conhecidos como *coletores de lixo móveis* (moving GCs). Quando uma variável é movida, todos os ponteiros que

armazenam o endereço da localidade antiga devem ser atualizados para apontar para o novo endereço. Do ponto de vista do coletor de lixo, um **unsafe.Pointer** é um ponteiro e, assim, seu valor deve mudar à medida que a variável se mover, mas um **uintptr** é apenas um número, portanto, seu valor não deve mudar. O código incorreto anterior *oculta um ponteiro* do coletor de lixo na variável **tmp**, que não é um ponteiro. Quando a segunda instrução executar, a variável **x** poderia ter sido movida e o número em **tmp** não seria mais o endereço **&x.b**. A terceira instrução sobrescreve uma posição arbitrária de memória com o valor 42.

Há inúmeras variações patológicas desse tema. Após a instrução a seguir ter sido executada:

```
pT := uintptr(unsafe.Pointer(new(T))) // NOTA: errado!
```

não há nenhum ponteiro que se refira à variável criada por **new**, portanto, o coletor de lixo pode reciclar sua área de armazenamento quando essa instrução for concluída; depois disso, **pT** contém o endereço em que a variável estava mas não está mais.

Nenhuma implementação de Go atualmente usa um coletor de lixo móvel (embora futuras implementações possam fazê-lo), mas isso não é motivo para ser complacente: versões atuais de Go movem *algumas* variáveis na memória. Lembre-se que, segundo a seção 5.2, as pilhas das gorrotina crescem conforme a necessidade. Quando isso acontece, todas as variáveis na antiga pilha podem ser realocadas para uma nova pilha maior; portanto, não podemos contar que o valor numérico do endereço de uma variável vá permanecer inalterado durante seu tempo de vida.

Quando escrevemos este livro, havia poucas orientações claras sobre aquilo com que os programadores Go poderiam contar após uma conversão de **unsafe.Pointer** para **uintptr** (veja a issue 7192 de Go). Assim, recomendamos fortemente que você pressuponha o mínimo. Trate todos os valores de **uintptr** como se contivessem o endereço *anterior* de uma variável e minimize o número de operações entre converter um **unsafe.Pointer** para um **uintptr** e usar esse **uintptr**. Em nosso primeiro exemplo anterior, todas as três operações – conversão para um

`uintptr`, soma do offset do campo e a conversão de volta – apareceram em uma única expressão.

Quando chamar uma função de biblioteca que devolva um `uintptr`, como as funções a seguir do pacote `reflect`, o resultado deve ser imediatamente convertido em um `unsafe.Pointer` a fim de garantir que ele continue a apontar para a mesma variável.

```
package reflect

func (Value) Pointer() uintptr
func (Value) UnsafeAddr() uintptr
func (Value) InterfaceData() [2]uintptr // (índice 1)
```

13.3 Exemplo: Equivalência profunda

A função `DeepEqual` do pacote `reflect` informa se dois valores são “profundamente” iguais (deeply equal). `DeepEqual` compara valores básicos como faria o operador embutido `==`; para valores compostos, ela os percorre recursivamente, comparando elementos correspondentes. Pelo fato de funcionar para qualquer par de valores, mesmo para aqueles que não são comparáveis com `==`, ela tem uso disseminado em testes. O teste a seguir usa `DeepEqual` para comparar dois valores `[]string`:

```
func TestSplit(t *testing.T) {
    got := strings.Split("a:b:c", ":")
    want := []string{"a", "b", "c"};
    if !reflect.DeepEqual(got, want) { /* ... */ }
}
```

Embora `DeepEqual` seja conveniente, suas distinções podem parecer arbitrárias. Por exemplo, ela não considera um mapa `nil` igual a um mapa vazio não `nil`, nem uma fatia `nil` igual a uma fatia vazia não `nil`:

```
var a, b []string = nil, []string{}
fmt.Println(reflect.DeepEqual(a, b)) // "false"

var c, d map[string]int = nil, make(map[string]int)
fmt.Println(reflect.DeepEqual(c, d)) // "false"
```

Nesta seção, definiremos uma função `Equal` que compara valores arbitrários. Como `DeepEqual`, ela compara fatias e mapas de acordo com

seus elementos, mas, diferentemente de **DeepEqual**, considera uma fatia (ou mapa) nil igual a uma fatia vazia (ou mapa vazio) não nil. A recursão básica pelos argumentos pode ser feita com reflexão, usando uma abordagem semelhante àquela usada no programa **Display** que vimos na seção 12.3. Como sempre, definimos uma função não exportada **equal** para a recursão. Não se preocupe ainda com o parâmetro **seen**. Para cada par de valores **x** e **y** a serem comparados, **equal** verifica se ambos são válidos (ou nenhum) e confere se são do mesmo tipo. O resultado da função é definido por um conjunto de casos de switch que comparam dois valores de mesmo tipo. Por questões de espaço, omitimos vários casos, pois, a essa altura, você já deve ter familiaridade com o padrão.

gopl.io/ch13/equal

```
func equal(x, y reflect.Value, seen map[comparison]bool) bool {
    if !x.IsValid() || !y.IsValid() {
        return x.IsValid() == y.IsValid()
    }
    if x.Type() != y.Type() {
        return false
    }
    // ...verificação de ciclo omitida (mostrada mais adiante)...
    switch x.Kind() {
    case reflect.Bool:
        return x.Bool() == y.Bool()
    case reflect.String:
        return x.String() == y.String()
    // ...casos numéricos omitidos por concisão...
    case reflect.Chan, reflect.UnsafePointer, reflect.Func:
        return x.Pointer() == y.Pointer()
    case reflect.Ptr, reflect.Interface:
        return equal(x.Elem(), y.Elem(), seen)
    case reflect.Array, reflect.Slice:
        if x.Len() != y.Len() {
            return false
        }
        for i := 0; i < x.Len(); i++ {
            if !equal(x.Index(i), y.Index(i), seen) {
```

```

        return false
    }
}
return true

// ...casos para estrutura e mapa omitidos por concisão...
}
panic("unreachable")
}

```

Como sempre, não expomos o uso de reflexão na API, portanto, a função exportada **Equal** deve chamar **reflect.ValueOf** em seus argumentos:

```

// Equal informa se x e y são profundamente iguais.
func Equal(x, y interface{}) bool {
    seen := make(map[comparison]bool)
    return equal(reflect.ValueOf(x), reflect.ValueOf(y), seen)
}

type comparison struct {
    x, y unsafe.Pointer
    t reflect.Type
}

```

Para garantir que o algoritmo termine, mesmo para estruturas de dados cíclicas, ele deve registrar quais pares de variáveis já foram comparados e evitar compará-los uma segunda vez. **Equal** aloca um conjunto de estruturas **comparison**, em que cada uma armazena o endereço de duas variáveis (representadas como valores de **unsafe.Pointer**) e o tipo da comparação. Precisamos registrar o tipo além dos endereços porque variáveis diferentes podem ter o mesmo endereço. Por exemplo, se **x** e **y** forem ambos arrays, **x** e **x[0]** têm o mesmo endereço, assim como **y** e **y[0]**, e é importante distinguir se comparamos **x** e **y** ou **x[0]** e **y[0]**.

Depois que **equal** determinou que seus argumentos são do mesmo tipo e antes de executar o switch, ela verifica se está comparando duas variáveis que já foram vistas; em caso afirmativo, a recursão é encerrada.

```

// verificação de ciclo
if x.CanAddr() && y.CanAddr() {
    xptr := unsafe.Pointer(x.UnsafeAddr())
    yptr := unsafe.Pointer(y.UnsafeAddr())
    if xptr == yptr {
        return true // referências idênticas
    }
}

```

```

    }
    c := comparison{xptr, yptr, x.Type()}
    if seen[c] {
        return true // já visto
    }
    seen[c] = true
}

```

Eis a nossa função **Equal** em ação:

```

fmt.Println(Equal([]int{1, 2, 3}, []int{1, 2, 3})) // "true"
fmt.Println(Equal([]string{"foo"}, []string{"bar"})) // "false"
fmt.Println(Equal([]string(nil), []string{})) // "true"
fmt.Println(Equal(map[string]int(nil), map[string]int{})) // "true"

```

Ela funciona até mesmo com entradas cíclicas semelhantes àquela que fez a função **Display** da seção 12.3 ficar presa em um loop:

```

// Listas ligadas circulares a -> b -> a e c -> c.
type link struct {
    value string
    tail  *link
}
a, b, c := &link{value: "a"}, &link{value: "b"}, &link{value: "c"}
a.tail, b.tail, c.tail = b, a, c
fmt.Println(Equal(a, a)) // "true"
fmt.Println(Equal(b, b)) // "true"
fmt.Println(Equal(c, c)) // "true"
fmt.Println(Equal(a, b)) // "false"
fmt.Println(Equal(a, c)) // "false"

```

Exercício 13.1: Defina uma função de comparação profunda que considere números (de qualquer tipo) iguais se eles se diferenciarem em menos de uma parte em um bilhão.

Exercício 13.2: Escreva uma função que informe se seu argumento é uma estrutura de dados cíclica.

13.4 Chamando código C com cgo

Um programa pode precisar usar um driver de hardware implementado em C, consultar um banco de dados embarcado implementado em C++ ou usar algumas rotinas de álgebra linear implementadas em Fortran. Há

muito tempo C tem sido a língua franca de programação, portanto, muitos pacotes para uso disseminado exportam uma API compatível com C, independentemente da linguagem de sua implementação.

Nesta seção, desenvolveremos um programa simples de compactação de dados que usa **cgo**, uma ferramenta que cria vínculos (bindings) de Go com funções C. Esses tipos de ferramenta são chamados de FFIs (Foreign-Function Interfaces, ou Interfaces de funções estrangeiras), e **cgo** não é a única ferramenta para programas Go. SWIG ([swig.org](http://www.swig.org)) é outra ferramenta; ela oferece funcionalidades mais complexas para integração com classes C++, mas não a mostraremos aqui.

A subárvore **compress/...** da biblioteca-padrão oferece compactadores e descompactadores para algoritmos populares de compactação, incluindo LZW (usado pelo comando **compress** do Unix) e DEFLATE (usado pelo comando GNU **gzip**). As APIs desses pacotes variam um pouco quanto aos detalhes, mas todas oferecem um wrapper para um **io.Writer** que compacta os dados escritos nele e um wrapper para um **io.Reader** que descompacta os dados lidos dele. Por exemplo:

```
package gzip // compress/gzip

func NewWriter(w io.Writer) io.WriteCloser
func NewReader(r io.Reader) (io.ReadCloser, error)
```

O algoritmo bzip2, baseado na elegante transformada de Burrows-Wheeler, executa mais lentamente que gzip, mas produz uma compactação significativamente melhor. O pacote **compress/bzip2** oferece um descompactador para bzip2, mas hoje em dia o pacote não tem nenhum compactador. Criar um do zero é uma tarefa substancial, mas há uma implementação de código aberto em C bem documentada e de alto desempenho: o pacote **libbzip2** de bzip.org.

Se a biblioteca C fosse pequena, simplesmente a portaríamos para Go puro e, se seu desempenho não fosse crítico para nossos propósitos, seria melhor chamar um programa C como um subprocesso auxiliar usando o pacote **os/exec**. Nos casos em que você precisa usar uma biblioteca complexa, de desempenho crítico, com uma API C restrita, é que pode

fazer sentido encapsulá-la usando **cgo**. No restante deste capítulo, trabalharemos com um exemplo.

Do pacote C **libbzip2**, precisamos do tipo estrutura **bz_stream**, que armazena os buffers de entrada e de saída, e de três funções C: **BZ2_bzCompressInit**, que aloca os buffers de stream, **BZ2_bzCompress**, que compacta dados do buffer de entrada para o buffer de saída e **BZ2_bzCompressEnd**, que libera os buffers. (Não se preocupe com o funcionamento do pacote **libbzip2**; o propósito desse exemplo é mostrar como as partes se encaixam.)

Chamaremos as funções C **BZ2_bzCompressInit** e **BZ2_bzCompressEnd** diretamente de Go, mas para **BZ2_bzCompress** definiremos uma função wrapper em C para mostrar como isso é feito. O arquivo-fonte em C a seguir está junto do código Go em nosso pacote:

gopl.io/ch13/bzip

```
/* Este é o arquivo gopl.io/ch13/bzip/bzip2.c,          */
/* um wrapper simples para libbzip2 adequado para cgo. */
#include <bzlib.h>

int bz2compress(bz_stream *s, int action,
                char *in, unsigned *inlen, char *out, unsigned *outlen) {
    s->next_in = in;
    s->avail_in = *inlen;
    s->next_out = out;
    s->avail_out = *outlen;
    int r = BZ2_bzCompress(s, action);
    *inlen -= s->avail_in;
    *outlen -= s->avail_out;
    s->next_in = s->next_out = NULL;1
    return r;
}
```

Vamos agora nos voltar para o código Go, cuja primeira parte está mostrada a seguir. A declaração **import "C"** é especial. Não existe um pacote **C**, mas essa importação faz **go build** pré-processar o arquivo usando a ferramenta **cgo** antes que o compilador Go o veja.

```
// O pacote bzip oferece um writer que usa compactação bzip2 (bzip.org).
package bzip
```

```

/*
#cgo CFLAGS: -I/usr/include
#cgo LDFLAGS: -L/usr/lib lbz2
#include <bzlib.h>

#include <stdlib.h>
bz_stream* bz2alloc() { return calloc(1, sizeof(bz_stream)); }
int bz2compress(bz_stream *s, int action,
                char *in, unsigned *inlen, char *out, unsigned *outlen);
void bz2free(bz_stream* s) { free(s); }
*/
import "C"

import (
    "io"
    "unsafe"
)

type writer struct {
    w      io.Writer // stream de saída subjacente
    stream *C.bz_stream
    outbuf [64 * 1024]byte
}

// NewWriter devolve um writer para streams compactados com bzip2.
func NewWriter(out io.Writer) io.WriteCloser {
    const blockSize = 9
    const verbosity = 0
    const workFactor = 30
    w := &writer{w: out, stream: C.bz2alloc()}
    C.BZ2_bzCompressInit(w.stream, blockSize, verbosity, workFactor)
    return w
}

```

Durante o pré-processamento, **cgo** gera um pacote temporário que contém declarações em Go correspondentes a todas as funções e tipos C usados pelo arquivo, como **C.bz_stream** e **C.BZ2_bzCompressInit**. A ferramenta **cgo** descobre esses tipos chamando o compilador C de maneira especial para processar o conteúdo do comentário que antecede a declaração de importação.

O comentário também pode conter diretivas **#cgo** que especificam opções extras para a cadeia de ferramentas C. Os valores **CFLAGS** e **LDFLAGS** contribuem com argumentos extras para os comandos do compilador e

do linker, de modo que eles possam localizar o arquivo de header **bzlib.h** e o arquivo compactado de biblioteca **libbz2.a**. O exemplo supõe que eles estão instalados abaixo de **/usr** em seu sistema. Talvez você precise alterar ou apagar essas flags em sua instalação.

NewWriter faz uma chamada para a função C **BZ2_bzCompressInit** a fim de inicializar os buffers para o stream. O tipo **writer** inclui outro buffer que será usado para drenar o buffer de saída do descompactador.

O método **Write** mostrado a seguir passa o **data** descompactado para o compactador, chamando a função **bz2compress** em um loop até todos os dados terem sido consumidos. Observe que o programa Go pode acessar tipos C como **bz_stream**, **char** e **uint**, funções C como **bz2compress** e até mesmo macros de pré-processador do tipo objeto em C, como **BZ_RUN**, tudo por meio da notação **C.x**. O tipo **C.uint** é diferente do tipo **uint** de Go, mesmo que ambos tenham o mesmo tamanho.

```
func (w *writer) Write(data []byte) (int, error) {
    if w.stream == nil {
        panic("closed")
    }
    var total int // bytes descompactados escritos
    for len(data) > 0 {
        inlen, outlen := C.uint(len(data)), C.uint(cap(w.outbuf))
        C.bz2compress(w.stream, C.BZ_RUN,
            (*C.char)(unsafe.Pointer(&data[0])), &inlen,
            (*C.char)(unsafe.Pointer(&w.outbuf)), &outlen)
        total += int(inlen)
        data = data[inlen:]
        if _, err := w.w.Write(w.outbuf[:outlen]); err != nil {
            return total, err
        }
    }
    return total, nil
}
```

Cada iteração do loop passa para **bz2compress** o endereço e o tamanho da parte restante de **data**, e o endereço e a capacidade de **w.outbuf**. As duas variáveis de tamanho são passadas pelos seus endereços, e não pelos seus valores, de modo que a função C possa atualizá-las e informar

quantos dados descompactados foram consumidos e quantos foram produzidos. Cada porção de dados compactados é então escrita no `io.Writer` subjacente.

O método `Close` tem uma estrutura semelhante a `Write` e usa um loop para descarregar qualquer dado compactado restante no buffer de saída do stream.

```
// Close descarrega os dados compactados e fecha o stream.
// Ele não fecha o io.Writer subjacente.
func (w *writer) Close() error {
    if w.stream == nil {
        panic("closed")
    }
    defer func() {
        C.BZ2_bzCompressEnd(w.stream)
        C.bz2free(w.stream)
        w.stream = nil
    }()
    for {
        inlen, outlen := C.uint(0), C.uint(cap(w.outbuf))
        r := C.bz2compress(w.stream, C.BZ_FINISH, nil, &inlen,
            (*C.char)(unsafe.Pointer(&w.outbuf)), &outlen)
        if _, err := w.w.Write(w.outbuf[:outlen]); err != nil {
            return err
        }
        if r == C.BZ_STREAM_END {
            return nil
        }
    }
}
```

Quando terminar, `Close` fecha `C.BZ2_bzCompressEnd` para liberar os buffers de stream, usando `defer` para garantir que isso aconteça em todos os caminhos de retorno³. A essa altura, o ponteiro `w.stream` não é mais seguro para desreferenciar. Por garantia, nós o definimos com `nil` e acrescentamos verificações explícitas para `nil` em cada método, de modo que o programa gera pânico se o usuário chamar um método após `Close` por engano.

Não só `writer` é inseguro para concorrência como também chamadas

concorrentes a **Close** e a **Write** fazem o programa falhar no código em C. Essa correção será feita no exercício 13.3.

O programa a seguir, **bzipper**, é um comando de compactação bzip2 que usa nosso novo pacote. Ele se comporta como o comando **bzip2** presente em muitos sistemas Unix.

gopl.io/ch13/bzipper

```
// Bzipper lê a entrada, compacta-a com bzip2 e escreve os dados na saída.
package main

import (
    "io"
    "log"
    "os"

    "gopl.io/ch13/bzip"
)

func main() {
    w := bzip.NewWriter(os.Stdout)
    if _, err := io.Copy(w, os.Stdin); err != nil {
        log.Fatalf("bzipper: %v\n", err)
    }
    if err := w.Close(); err != nil {
        log.Fatalf("bzipper: close: %v\n", err)
    }
}
```

Na sessão a seguir, usamos **bzipper** para compactar **/usr/share/dict/words**, o dicionário do sistema, de 938.848 bytes para 335.405 bytes – aproximadamente um terço de seu tamanho original –, e então o descompactamos com o comando **bunzip2** do sistema. O hash SHA256 é o mesmo antes e depois, o que nos dá confiança de que o compactador está funcionando corretamente. (Se você não tiver **sha256sum** em seu sistema, use sua solução do exercício 4.2.)

```
$ go build gopl.io/ch13/bzipper
$ wc -c < /usr/share/dict/words
938848
$ sha256sum < /usr/share/dict/words
126a4ef38493313edc50b86f90dfdaf7c59ec6c948451eac228f2f3a8ab1a6ed -
$ ./bzipper < /usr/share/dict/words | wc -c
335405
$ ./bzipper < /usr/share/dict/words | bunzip2 | sha256sum
126a4ef38493313edc50b86f90dfdaf7c59ec6c948451eac228f2f3a8ab1a6ed -
```

Mostramos como ligar uma biblioteca C em um programa Go. Indo na outra direção, também é possível compilar um programa Go como um arquivo estático que pode ser ligado em um programa C ou como uma biblioteca compartilhada que pode ser dinamicamente carregada por um programa C. Mal tocamos a superfície de **cgo** aqui, e há muito mais a dizer sobre gerenciamento de memória, ponteiros, callbacks, tratamento de sinais, strings, **errno**, finalizadores e o relacionamento entre gorrotinas e threads do sistema operacional – e boa parte desses temas são bastante delicados. Em particular, as regras para passar ponteiros corretamente de Go para C e vice-versa são complexas, por motivos semelhantes àqueles discutidos na seção 13.2, e ainda não foram especificadas de forma definitiva⁴. Para outras leituras, comece em <https://golang.org/cmd/cgo>.

Exercício 13.3: Use **sync.Mutex** para deixar **bzip.writer** seguro para uso concorrente por várias gorrotinas.

Exercício 13.4: Dependendo de bibliotecas C tem suas desvantagens. Ofereça uma implementação alternativa puramente em Go para **bzip.NewWriter** que use o pacote **os/exec** para executar **/bin/bzip2** como um subprocesso.

13.5 Outra advertência

Terminamos o capítulo anterior com um aviso sobre as desvantagens da interface de reflexão. Aquele aviso aplica-se com mais força ainda ao pacote **unsafe** descrito neste capítulo.

Linguagens de alto nível isolam programas e programadores, não só de

particularidades misteriosas de conjuntos de instruções de CPUs específicas, mas também de dependências de irrelevâncias como o local em que uma variável reside na memória, o tamanho de um tipo de dado, os detalhes do layout de estruturas e toda uma variedade de outros detalhes de implementação. Por causa dessa camada de isolamento, é possível escrever programas que sejam seguros e robustos e que executarão em qualquer sistema operacional, sem alterações.

O pacote **unsafe** permite aos programadores furar o isolamento e usar alguns recursos fundamentais que, de outra forma, seriam inacessíveis, ou, quem sabe, conseguir um desempenho melhor. O custo geralmente incide sobre a portabilidade e a segurança, portanto, uma pessoa deve usar **unsafe** por sua conta e risco. Nosso conselho sobre como e quando usar **unsafe** tem paralelo nos comentários de Knuth sobre otimização prematura, que citamos na seção 11.5. A maioria dos programadores jamais precisará usar **unsafe**. Apesar disso, ocasionalmente, haverá situações em que algumas partes críticas do código podem ser melhor implementadas usando **unsafe**. Se estudos e medidas cuidadosos indicarem que **unsafe** de fato é a melhor abordagem, restrinja-o à menor região possível, de modo que a maior parte do programa não tome conhecimento de seu uso.

Por enquanto, coloque os dois últimos capítulos no fundo de sua mente. Escreva alguns programas Go substanciais. Evite **reflect** e **unsafe**; volte a esses capítulos somente se precisar.

Enquanto isso, tenha uma feliz programação em Go. Esperamos que você goste de escrever código Go tanto quanto nós.

¹ Nota do Revisor Técnico da Tradução: linha acrescentada para resolver problema descrito no item p.362 da errata do original, relativo ao uso de ponteiros compartilhados entre C e Go. Ver nota seguinte.

² Nota do Revisor Técnico da Tradução: este `#include`, e as funções `bz2alloc` e `bz2free` em linguagem C foram acrescentados para fazer uso correto da estrutura `bz_stream` compartilhada entre C e Go, como descrito no item p.362 da errata do original. Resumindo, a alocação e liberação de `bz_stream` deve ser feitas pelas funções em C, e não pelo código Go como estava antes. O código completo corrigido está em: <https://github.com/adonovan/gopl.io/blob/master/ch13/bzip>

- 3 Nota do Revisor Técnico da Tradução: ainda na função adiada com defer, a chamada à função em C `bz2free(w.stream)` libera explicitamente a estrutura `bz_stream` alocada por `bz2alloc()` em `NewWriter`. Isso completa a solução do problema descrito no item p.362 da errata da 3ª impressão do original.
- 4 Nota do Revisor Técnico da Tradução: após o lançamento da 1ª edição do original deste livro, as regras foram formalizadas no documento “Rules for passing pointers between Go and C”, disponível em <https://github.com/golang/proposal/blob/master/design/12416-cgo-pointers.md>



Your gateway to knowledge and culture. Accessible for everyone.



z-library.se

singlelogin.re

go-to-zlibrary.se

single-login.ru



[Official Telegram channel](#)



[Z-Access](#)



<https://wikipedia.org/wiki/Z-Library>